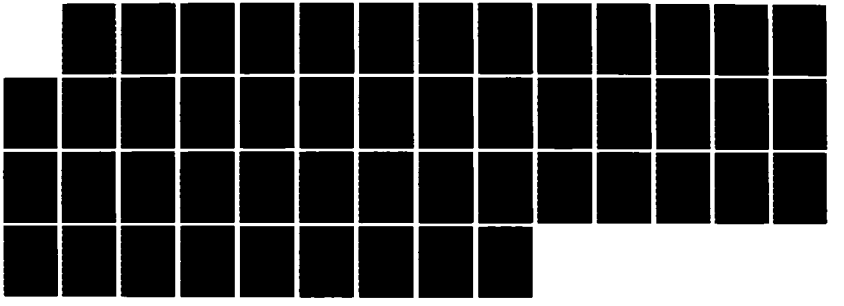
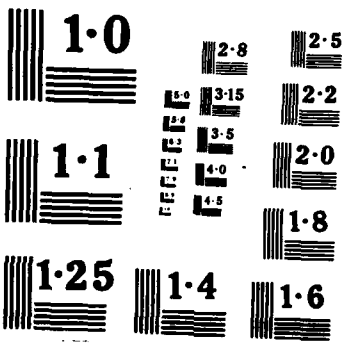
AD-A170 350 HPC (HIERARCHIAL PROCESS COMPOSITION): A MODEL OF STRUCTURE AND CHANGE IN..(U) ROCHESTER UNIV NY DEPT OF COMPUTER SCIENCE T J LEBLANC ET AL. MAY 85 TR-153 1/1

UNCLASSIFIED DACA76-85-C-0001 F/G 9/2 NL



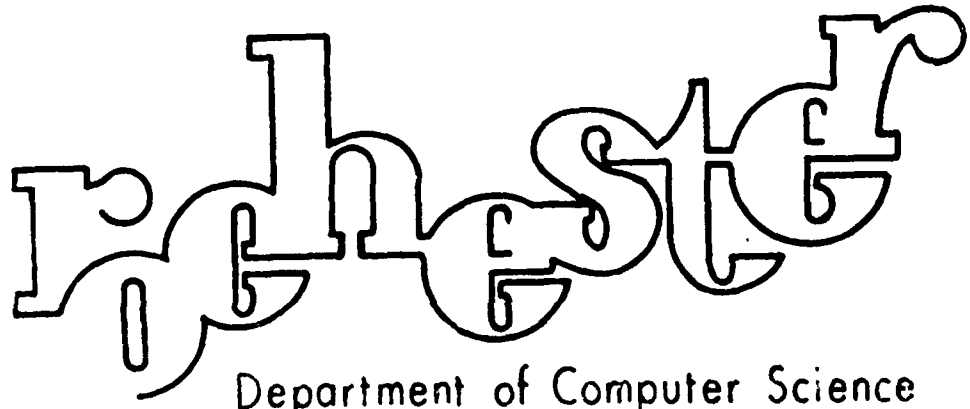

1·0
2·8
2·5
3·15
2·2
3·5
2·0
1·1
4·0
4·5
1·8
1·25
1·4
1·6

(12)

HPC: A Model of Structure and Change in Distributed Systems

Thomas J. LeBlanc and Stuart A. Friedberg
Computer Science Department
The University of Rochester
Rochester, New York 14627

TR 153
May 1985



Department of Computer Science
University of Rochester
Rochester, New York 14627

HPC: A Model of Structure and Change in Distributed Systems

Thomas J. LeBlanc and Stuart A. Friedberg
Computer Science Department
The University of Rochester
Rochester, New York 14627

TR 153
May 1985

Abstract

Distributed systems must provide certain fundamental facilities including communication, protection, resource management, reliability, and process (computation) abstraction. Current designs for distributed systems tend to focus on only one of these issues; support for multiprocess structures has been especially neglected. The HPC model, an object-oriented model of interprocess relationships for distributed systems, addresses all of these fundamental services. The major novelties of HPC lie in the extension of the process abstraction to collections of processes and the provision of a rich set of structuring mechanisms for distributed computations. An important aspect of the model is that it results in the ability to maintain and exploit execution context for managing processes in a distributed computation. In this paper we describe the HPC model, show how the model can be used to build system-level services, and discuss the implementation of an HPC kernel.

This research was supported by National Science Foundation grant number DCR-8320136 and DARPA/ETL grant number DACA76-85-C-0001.

AD-A170 350

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER TR 153	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) HPC: A Model of Structure and Change in Distributed Systems		5. TYPE OF REPORT & PERIOD COVERED technical report 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Thomas J. LeBlanc and Stuart A. Friedberg		8. CONTRACT OR GRANT NUMBER(s) DACA 76-85-C-0001
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Science Department University of Rochester Rochester, NY 14627		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE May 1985 13. NUMBER OF PAGES 35
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) U.S. Army Engineer Topographic Laboratories Attn: ETL-RI (Dr. Leighty) Fort Belvoir, VA 22060		15. SECURITY CLASS. (of this report) unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this document is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

None.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

distributed systems, process structuring, interprocess communication, protection, abstraction

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Distributed systems must provide certain fundamental facilities including communication, protection, resource management, reliability, and process (computation) abstraction. Current designs for distributed systems tend to focus on only one of these issues; support for multiprocess structures has been especially neglected. The HPC model, an object-oriented model of interprocess

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

relationships for distributed systems, addresses all of these fundamental services. The major novelties of HPC lie in the extension of the process abstraction to collections of processes and the provision of a rich set of structuring mechanisms for distributed computations. An important aspect of the model is that it results in the ability to maintain and exploit execution context for managing processes in a distributed computation. In this paper we describe the HPC model, show how the model can be used to build system-level services, and then discuss the implementation of an HPC kernel.

1. Introduction

Distributed systems must provide certain fundamental services including communication, protection, resource management, reliability, and process (computation) abstraction. Current designs for distributed systems tend to focus on only one of these issues. For example, many systems have focused primarily on communication: How do processes communicate? When do they communicate? What assumptions about communication are necessary? Of the basic services, support for multiprocess structures has been especially neglected.

Distributed systems tend to be complex, with many distinct components. The computation elements and the communication paths among them comprise the structure of a distributed system. Designers of distributed applications will choose a wide variety of logical structures when designing their systems. No complex system can be built without such structures, yet most structure disappears in the implementation of a distributed application on top of a distributed operating system. One may typically expect the implementation to contain a mass of processes with no indication of how they are related by function or communication. What has been lacking is a detailed exploration of the underlying structure of distributed systems: How are objects structured? How do user programs interact with system services? What support, beyond the need to provide communication, is necessary for long-running distributed computations?

This work has been motivated by the observation that most operating systems lack support for structuring dynamic relationships between processes. Whatever structure is provided is usually implicit and cannot be exploited. For example, few systems allow for the regulated composition of processes. It is difficult or impossible to create complex relationships between computational units constructed with abstract components. In addition, operating systems do not maintain a global context for an executing process. Since processes cannot exploit the context in which they execute, error recovery must be based solely on local context, even when global context may provide a better recovery strategy.

Distributed operating systems are not the only systems that provide little or no recognition of interprocess relationships. It is unfair to criticize distributed systems for failing to provide sophistication absent in conventional systems. However, these issues are especially critical for distributed systems because they can involve hundreds of machines and thousands of services. The amount of software and the number of users involved can be very large. The increasingly blurred distinction between system services and clients in distributed systems suggests that many programmers will require greater interaction with system-related software, for example, to provide application-specific recovery mechanisms. This interaction can be made simpler and less error-prone if the operating system is structured appropriately. In addition, distributed programming is fundamentally more difficult than writing programs for a single processor, especially when a program must be able to compensate for machine failures. Software support for a programming methodology that facilitates distributed programming is especially important. Finally, interprocess relationships are more important in distributed systems because applications are expected to use multiple processes as a matter of course.

Hierarchical Process Composition (HPC) is an object-oriented model of interprocess relationships designed to address these issues. The major novelties of HPC lie in the extension of the process abstraction to collections of processes and the provision of a rich set of structuring mechanisms for distributed computations. The fundamental tool in the HPC model is *abstraction*, which, in various forms, is used to provide object definition, protection, and composition. Structuring forms are defined that allow a program to create protection domains that permit, yet limit, structural modification of objects. An important aspect of the model is that it results in the ability to maintain and exploit execution context for managing processes in a distributed computation.

In the HPC object model, naming, protection, resource management, data, procedures, processes, and communication are all characterized in terms of discrete objects. There are no operations on passive objects. All computational interactions between objects take place by sending and receiving explicit messages, a view that distinguishes HPC from the abstract data type object model.[1,2] In addition, the structural relationships between objects are explicitly managed in HPC, unlike continuation-style binding found in actors[3] and Smalltalk-80.[4]

An inspirational force behind this work is the simplicity of process structures in the Unix† operating system. Process families in Unix are tree-structured: each process has a single parent and may create many children. Unfortunately, Unix does not really support the process tree structure. For example, a process can communicate via signals with any other process, without regard to the process hierarchy. In addition, the process tree structure is difficult to exploit; children do not naturally know their parent process and parents cannot directly disown children. *Process groups*, an addition made in Berkeley Unix,[5] among others, is an attempt to partially rectify the problem. This addition to the process hierarchy allows signals to be sent to a group of processes, but does not provide a general solution for the management of process subtrees.

Unix processes communicate via pipes, a mechanism whereby the output of one program becomes the input of another program. In effect, a pipe is the communication channel that links the output file descriptor of one process to the input file descriptor of another. This communication channel acts as a speed-matched buffer, preventing a process from sending data faster than another can receive it. Multiple processes can be linked using pipes to form pipeline structures. The primary disadvantage of the Unix pipe mechanism is that the relationship between the processes is poorly expressed. There is no mechanism for building a complex sequence of communicating processes into a pipeline structure and binding a name to that structure, such that many instances can be created, manipulated, and composed. That is, pipes do not allow arbitrary composition. In addition, pipes are implemented using an I/O stream interposed between two processes that use it to communicate; no higher authority understands and oversees this relationship. The effect is that, in Unix, the failure of any segment in a pipeline causes the entire pipeline to be aborted since, without additional context, there is little information available to guide less drastic recovery mechanisms.

Communication in the HPC model is similar in flavor to the message-based operating systems Demos[6] and Arachne[7] (formerly Roscoe). Both of these systems use *links* to connect communicating processes. A link combines the notions of communication path and capability. The holder of a link may send messages to the owner, i.e., all links are uni-directional. The holder can duplicate the link or give it away subject to restrictions specified by the owner, but the same receiving process is always implicitly associated with the link. A sending process names a link only, not the destination process.

A *switchboard task* or *resource manager* is used to establish links between two unrelated processes. Each new process is created with a link to the switchboard that allows a process to request links to other processes. Control information, in the form of asynchronous notification, may be generated by the kernel for the owner of a link when the link is copied or undergoes a status change.

In both of these systems, a user process that wants to send messages to the owner of a link must explicitly get access to the appropriate link, usually via the switchboard process. Thus, user processes must explicitly manipulate the connections to their interfaces. In HPC, a process always communicates by sending messages to an internal interface. Connections between two interfaces, necessary for

†Unix is a trademark of Bell Laboratories

communication to take place between processes, are transparent to the sending and receiving processes. A standard system component that connects and disconnects the appropriate interfaces can be included as part of each program. Autonomy is not sacrificed because individual processes still maintain complete control over their own internal interfaces.

HPC interfaces provide more structure (typing or interpretation for messages) than IPC sockets[8] and less structure than Accent ports.[9] Unlike both sockets and Accent ports, HPC interfaces are permanently associated with a specific process. An HPC interface may be thought of as a named entry point for a concurrent task or process.

In many message-based operating systems, including Demos and Arachne, a single message between processes uses the same mechanisms and receives the same level of support as a series of messages between processes with a long-term relationship. This can be justified in an environment in which most communication is short in duration. However, for many distributed *computations*, communication connections can be long-lived. What is lacking is a mechanism for codifying long-term relationships, so that explicit support can be provided consistent with the needs of the processes involved. One of the assumptions underlying the HPC model is that the granularity of process interactions in distributed systems can vary greatly depending on the relationship between the processes involved. Therefore, the model provides structuring mechanisms for making long-term relationships explicit.

Much of the structure of HPC is similar to that of Eden, [10,11] an object-based distributed computing environment developed at the University of Washington. Eden objects, called ejects, communicate via messages. The only connection with the outside world for an eject is defined by its invocations (*i.e.*, interfaces); the internal structure of an eject is the concern of its programmer alone. Ejects are built using processes communicating via monitors controlled by a single coordinator process.

There are three main differences between Eden and HPC: (1) Eden provides invocation as the primary mode of communication; HPC uses asynchronous messages, (2) Eden uses capabilities to check access rights to invocations; HPC was explicitly designed to avoid the use of capabilities, and (3) Eden provides a mostly flat collection of objects; HPC provides a rich set of mechanisms for creating nested object structures.

There is a close relationship between the HPC model and the *activity* model also developed at the University of Rochester. The activity model defines a conceptual tool for describing the relationships between objects involved in the execution of a distributed task.[12] A single object may participate in many different activities and a single activity may be made up of numerous subactivities. Activity tags are used to identify the activity affiliation of data and messages. HPC began as an attempt to define, in detail, operating system support for the activity model. It quickly diverged, although previous work on activities continues to have important influence.

In section 2 of this paper, we present the HPC object model, introducing terminology used throughout the paper. Sections 3-6 provide further details of the model, describing the concept of operation domains for protection, structured interfaces for communication, how to maintain the consistency of an HPC system in the presence of partition, and HPC support for sharing. Section 7 shows how to build representative system-level services using the HPC model. In section 8, we discuss the implementation of an HPC kernel (currently under development), and summarize, in section 9, the motivation and rationale behind the HPC model.

2. An Overview of the HPC Object Model

HPC can be viewed as an extension of the process abstraction model. A set of communicating processes can be encapsulated to form an abstraction whose status within the system is equal to that of an ordinary process. Any commands that can be applied to a single process can be applied to the abstract object representing a set of processes. We have chosen hierarchical composition as the basic structuring mechanism for process collections.

2.1. Objects

The simplest type of object in HPC is the *process*, which consists of active state, communication interfaces, and a code segment. An *object* can be constructed from component parts by combining other previously created objects, communication *channels* between sub-objects, an *encapsulation shell*, and a set of *interfaces* to the external world. The behavior of a complex object is defined by the behavior of its component objects and the structure of their interface connections. A fundamental requirement of the model is that the semantics of external-world interfaces be the same for both complex and simple objects.

When an object is created, it is given a unique name. The name of an object is known by its creator and the object itself; these two objects may in turn pass the name on to other objects. The system commands that modify the structure of an object require that the name of the object be known. Sending a message to an object, however, does not require that the sender know the name of the object, only that the appropriate interface in the sender be bound to an interface in the receiver by a channel.

2.2. Interfaces and Channels

Communication between objects can only occur by sending messages through channels. A channel is a directed communication medium that allows a message to be sent asynchronously from one object to another. To the underlying system, messages are simply uninterpreted values. A channel does not provide queuing of messages or reliable delivery. Reliability, if necessary, must be provided by higher-level protocols.

An interface describes the type of message that can be transmitted by an object through a uni-directional channel. Each object must define an interface for each potential communication stream. Channels are bound to interfaces by run-time operations. At any point in time, some object interfaces may be bound to channels while others are not. Messages that are sent using an unbound interface are lost.

In order for two objects to communicate, the appropriate interfaces of each object must be connected to the same channel. The **connect** operation is used to create a channel joining two specified interfaces. The **disconnect** operation is used to remove the channel, terminating the connection. In this connection-based approach, the location and identity of partners in communication is transparent to the communicating objects, since each object is communicating through a locally named interface. This makes it easier for the underlying system to mask failures and provide process migration. In addition, the control aspects of communication (*i.e.*, with which other object does a particular object communicate) are separated from the communication itself and can be governed by a third party whose context includes both communicating parties.

There are three communication operations that can be performed on an interface: **send**, **receive**, and **select**. HPC is designed to model asynchronous, autonomous operations, hence, the **send** and **receive**

primitives are nonblocking. The **select** primitive is analogous to the Ada† select statement, which provides a form of blocking receive, eliminating the need to poll individual interfaces.

2.3. Encapsulation Shells

Many modern programming languages (*e.g.*, Modula-2, Ada) allow the programmer to encapsulate an abstract object within a module that presents a well-defined interface to the outside world. This has the advantage of separating the abstract behavior of the object from the implementation of the object. An HPC encapsulation shell is similar to a module in that it provides a structuring form for processes. A shell is an object, along with all its contents. Shells are designed to support the composition of objects. A composite object can be created by encapsulating a set of communicating objects within a shell that provides appropriate interfaces, thereby creating a "black box" that can be combined with other "black boxes" to form more complicated objects. A shell serves both as an abstraction mechanism and as a handle for structured objects.

Not all modules in programming languages are purely syntactic constructs (*e.g.*, interface modules or monitors in Modula). Similarly, not all shells are purely abstraction mechanisms. An *operation domain* is a shell created with a designated control component, called the *controller*, responsible for modifying and maintaining the internal structure of the objects within the domain. In particular, the controller is responsible for establishing communication connections

All shells form a strict tree-structured hierarchy. A new shell can be created around a collection of sibling objects using the **enclose** operation. The number and types of the shell's interfaces, as well as the collection of objects it is to enclose, are given as arguments to **enclose**. The specified objects become the children of the newly created shell. Any previous connections between objects inside and outside the shell are removed. The **disclose** operation removes a shell and any channels connected to its interfaces. The children of the shell become children of the shell's parent.

A shell may enclose an arbitrarily complex collection of objects. The externally visible behavior of the resulting object is defined by the shell's communication interfaces. A shell can be transparent or opaque. The internal structure of an opaque shell, including internal channels and the number and type of sub-objects, is invisible outside the shell.

A shell has a name and a set of interfaces. When an object is created, the properties of its interfaces are fixed for the lifetime of the object. Each interface is visible on both sides of the shell and may display different properties on either side. Therefore, we distinguish the internal and external *views*. Associated with the external (internal) view are the interface properties visible from outside (inside) the shell. Neither view has any special status not held by the other. The internal view is used to connect a channel between an object within the shell and the shell itself. The external view is used to forward messages from the shell to other objects. Thus, some interfaces are associated with active processes, while others are used to forward messages between compatible interfaces in different shells. This does not imply, however, that the implementation must actually forward messages through a series of physical interfaces.

In any interaction with other objects, a complex object is indistinguishable from a simple object with the same interfaces. Commands that can only be applied to simple processes in other systems may be applied to these complex objects in HPC. For example, the execution of an object can be halted without regard to whether the object is a simple process or a complex object.

---

†Ada is a registered trademark of the U.S. Department of Defense

Figure 1 shows three HPC objects. Figure 1a contains an abstract object with two external interfaces. This object could be either a simple process or a complex object; there is no way to tell since the object is opaque. The only information available to the outside world is the name of the object and the names, number, and types of the object's external interfaces. In Figure 1b three objects are shown connected in a pipeline. Again, each object in the pipeline could be either a process or a complex object. We have not shown the direction of communication flow through the pipeline. Finally, in Figure 1c a complex object is shown with the same abstract behavior as the three objects in Figure 1b. This object, currently transparent and under control of the enclosing domain, can be given a controller, causing the object to become opaque and capable of internal modification to its structure. The next section explains how this is done.

3. Operation Domains and Controllers

The HPC model recognizes the need for dynamic relationships between processes. Communication channels can be dynamically established between processes and shells can be created and destroyed to represent the changing relationships between objects. These operations allow complex interprocess relationships to be created, but their unrestricted application could lead to chaos. HPC restricts the execution of structure modifying operations based on additional hierarchical structure, in the form of *operation domains.*

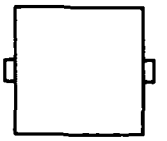
The terms *domain* and *principal* are taken from the security and protection literature. A domain is a collection of objects that have the same protection or access control. A principal is an agent that has privileged access to the members of a domain. In HPC, operation domains (or simply, domains) are delineated by a shell and privileged access amounts to the right to execute any of the structure modifying operations. The contents of a domain are everything inside the domain shell that is not inside another (nested) domain. In this way, operation domains form a coarse tree structure that is superimposed on the object hierarchy of HPC. Each domain has exactly one principal, called a *controller*, which is the only object that may modify or even examine the contents of the domain. Controllers are members of the domains they control and may not examine or affect the contents of other domains. Thus, shells that delineate a domain are opaque since no object can observe both sides. Shells that do not delineate a domain are transparent to the controller of the domain in which they reside. In this way, protection and access control to HPC objects and their connections are associated with the basic hierarchy.

The only objects that may be affected by more than one controller are shells that delineate operation domains. A domain's shell is the root object in its own hierarchy and a leaf object in the domain that encloses it. The controller of the internal domain is restricted to operations on the internal view of the shell and its interfaces, while the controller of the enclosing domain is restricted to operations on the external view.

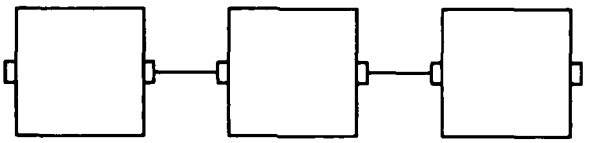
3.1. Controller Subtrees

HPC implements principals through the notion of a *controller subtree.* The root of a controller subtree may be identical with the root of the domain, in which case we call it a *level 0* controller, or it may be an immediate child of the root of the domain, in which case it is a *level 1* controller. In all cases the subtree may be of arbitrary complexity, from a single leaf process to a complex tree with nested domains within it.

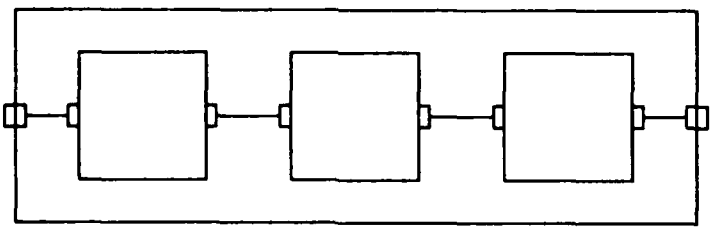
Implementing principals as controller subtrees rather than single processes makes it possible to distribute the control function over several sites. In addition, it allows us to use the same structuring tools in implementing complex control functions as in implementing applications. We do not have enough



(a)



(b)



(c)

Figure 1

experience to determine if this flexibility is necessary, however, this approach does have significant advantages.

A domain with a level 0 controller is a *self-controlling domain.* All its members have the same power to modify and maintain its internal structure. By definition, a primitive, or leaf, process is a self-controlling domain. Its internal structure is hidden from every other object in the HPC environment. Most existing distributed systems can be readily modelled by providing each user with a self-controlling domain where all user processes are immediate children of the domain. This allows all such processes to interact in arbitrary ways with processes owned by the same user; interaction with processes owned by other users must use the interface presented by the domain boundary.

Domains with level 1 controllers provide for the separation of control from computation and finer access control. The subobjects that are not members of the controller subtree have no need or even the capability to create new processes, alter their interconnections, or otherwise reconfigure the domain. These functions are reserved for the members of the controller subtree, which can be specialized for the control task. There is an analogy with the various *shell* programs of the Unix operating system, all of which specialize in creating new processes with the appropriate interconnections, monitoring their completion status, arbitrating between them for access to the user's terminal, and so forth. Each shell is a valuable tool, yet none performs any real computation. At the same time, user processes generally do not need to know anything about their interconnections or relationships with other processes.

In theory, there is no reason why the controller for a domain could not be arbitrarily located in the object tree. However, to simplify both the design and the implementation of the system, we have decided that a controller must be a subtree of its domain, and furthermore, that the subtree must begin at the root or one of its immediate children. It is easy to verify that a given process is authorized to carry out an operation on an object if the process, the object and their common ancestor are all in the same domain. The operations that create and destroy domains and controllers are easier to define and implement when the controller subtrees are not arbitrarily deep within their domains.

Primitive processes are ultimately the only objects that will directly attempt to execute any operations, since they are the implementation base of all complex objects. We must specify which primitive processes are empowered to act as controllers for a particular domain. That is, which processes are allowed to execute structural modification operations within a domain? For consistency and simplicity, we would like our specification to ensure these properties: (1) a process is empowered as a controller of at most one domain, (2) a process may be empowered as a controller only for its domain or the immediately enclosing domain, (3) a controller subtree may itself be a domain. and (4) a controller subtree may not be both a level 0 and a level 1 controller.

Assume we have a domain D and a controller subtree S. We cannot adopt the rule that all processes in S that are also in D are empowered as controllers because there are no processes in D; each process is in its own individual domain. If we adopt the rule that all processes in S are empowered as controllers of D, it would violate the rule that each controller has privileges for just one domain, since S may contain domains with their own internal controller subtrees. Therefore, we have adopted the rule that if S is a controller subtree for domain D, then a process P is empowered as a controller of domain D iff

(1) P is a member of S

(2) There is no object strictly between P and the root of S that is a controller subtree[†]

[†]A is strictly between B and C if C contains A, A contains B, and A, B and C are all distinct objects.

(3) There is no object strictly between P and the root of S that is a domain

These requirements are sufficient to ensure the desired properties.

Figure 2 illustrates nested domains and controllers. Shell S0 delineates a domain, D1, with controller C1. The structure of D1 may only be modified by a process within C1. However, C1 also delineates domain D2, whose controller is C2. Process P1 may modify D1, for example, by connecting Pi and Pj. Processes within C2 may not since each process can be a controller for at most one domain. Similarly, processes in C2 may modify D2, but P1 may not, since it is not in the controller subtree for D2.

3.2. The Role of the Controller

The processes of the controller subtree are solely privileged and responsible for issuing all the HPC commands that manipulate shells, interfaces, processes, and controllers. All the arguments to such a command must be members of the controller's domain. One consequence is that communication across a domain boundary is possible only with the cooperation of the controllers on both sides of the boundary.

The primary purpose of the controller is to maintain the health of objects within its domain and the relationships between those objects. Detection and recovery from failure is a major responsibility. Reconfiguration of the object to accommodate changes in load, unusual demands for service, or global changes, such as partition, is another task of the controller.

Beside routine health and maintenance, the controller holds all the mechanisms needed to debug at the level of HPC objects and their interactions. For example, the controller may interact with objects using the standard message communication primitives available to all objects, as well as the operations available to it alone. Using **send** and **receive** and its ability to reconnect objects in arbitrary ways, a controller can monitor all communication between two objects by establishing a communication path between the objects that passes through the controller's interfaces or the interfaces of a selected debugging module. Monitored communication would be transparent to the two objects involved.

The controller can establish a *control channel* with each of the objects within its domain for non-transparent interaction. The role of the control channel is, by convention, to send commands to the individual objects. Since the controller's ability to modify its domain is limited to a specific set of commands (*e.g.*, connect interfaces), this channel can be used to request that the object change state according to established conventions. For example, an object can be destroyed by its controller, but cannot be forced to enter a particular state by the controller. Thus, the control channel would be used for commands that require the consent of the affected object.

In the same vein, a *request channel* can be established between the controller and an object that provides the object with access to the services of the the controller, such as a request to make a specific connection. All requests that require a reply would involve both the request channel and the control channel. Objects do not have to define interfaces for a control or request channel, although it is expected that most long-lived objects will do so. Once created, short-lived computations with static connections will not usually require controller intervention.

3.3. Manipulating Processes

Process management in HPC is primarily connected with protection and secondarily with resource management. There is a tension in the HPC design between the desire to exploit the underlying topology of the distributed hardware and the desire to treat all objects in a uniform way. Since HPC may be implemented over heterogeneous sites, management of raw processes is necessarily a site-dependent

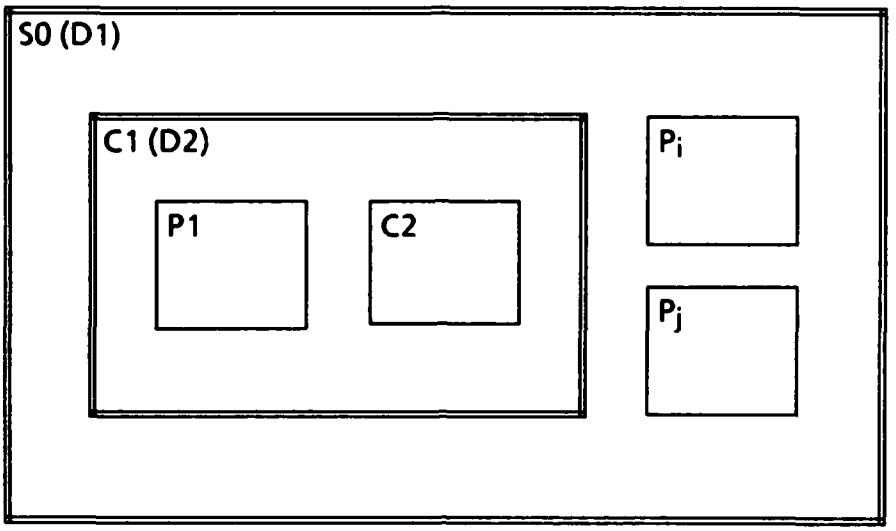


Figure 2

problem. We have decided to separate the resource management of process allocation and scheduling from the structure management of the HPC hierarchy. No one, not even controllers, ever observes a raw process within HPC. As with other resources, processes at a site or set of sites are managed through an HPC object. The details of identifying an executable image, controlling scheduling priorities, and so forth are all independent of the HPC model, as part of the private exchange of messages between two objects. This scheme allows HPC objects to manipulate site-dependent processes that exist entirely outside of the HPC framework.

Introducing new processes into the HPC world is of more interest. HPC provides a primitive operation, **animate**, that takes an empty shell, the identity of a process manager, and a manager-dependent process description. Successful animation creates a new process and makes it a new domain delineated by the shell. The HPC kernel remembers the association between the raw process and its domain so that HPC operations on the domain will be translated into the appropriate process management primitives on the raw process.

Animation is actually a bit more complicated than just described. The process description given as an argument identifies the HPC equivalent of an executable image. We considered allowing the image to describe a tree of objects, complete with controller subtree and initial connections, so that instantiating the image would produce an entire subtree within the shell provided. However, we chose a simpler mechanism that is capable of doing everything the apparently more powerful mechanism can. An image specifies a single process and whether it is to be instantiated as a level 0 or a level 1 controller. An ordinary process would be instantiated as a level 0 controller. In this case, a new self-controlling domain would be superimposed on the shell given to **animate**. In the other case, a domain would be superimposed on the shell, but a new subshell would be created within the given shell and the process would be instantiated within the subshell, not the given shell. The subshell would also be designated the controller subtree of the domain.

Instantiating a process as a level 1 controller inside the initial shell gives it an environment to instantiate other processes and build up an arbitrarily complex domain. The operations that manipulate controllers, discussed below, allow the initial controller to build up a more complex controller tree using the ordinary structure modifying operations and then transfer the designation of controller to the new tree.

When an object dies, its domain is destroyed, removing the internal structure and leaving the shell behind. Termination of a primitive process is treated as a death by HPC. A controller may execute a **die** operation on its domain, which will remove all internal structure before removing the domain boundary that protected the structure from the view of the enclosing domain. A controller may **kill** some member M of its domain, which is equivalent to forcing all the domains nested within M to execute a **die** and then removing any shells remaining within M.

3.4. Manipulating Controllers

Controller subtrees may be introduced through the **invest** operation. A controller may select some subtree S of its domain and invest a new controller C for that subtree. If S is the domain itself, this amounts to a transfer of control from the original controller to C. Otherwise, a new domain is created. If C was previously a self-controlling domain (excluding primitive processes), the old domain boundary of C is destroyed to assure that no process is a controller for more than one domain. If C represents a domain with a level 1 controller, it remains a domain and its controller is limited to modifications within C. Members of C that are not members of its controller subtree are empowered as controllers of S. This also ensures that each process is controlling at most one domain.

Complex objects can be created either top-down or bottom-up. A problem arises, however, if domains are constructed top-down. Once a controller is invested with a domain, the controller for the enclosing domain no longer has the ability to operate within the newly created domain. Thus, only objects capable of building their own internal structure can be constructed top-down within nested domains.

A controller may **abdicate** its power over a domain. When a controller abdicates, its domain shell becomes transparent and the contents of its domain become contents of the enclosing domain. A controller may **depose** a subdomain, which is equivalent to forcing the subdomain to abdicate.

When an object dies, there is no way for it to be resurrected, short of external intervention by the controller of the domain in which it resides. To ensure that all domains always have a controller, HPC treats the death of a controller subtree as an abdication. This takes place even though other objects within the domain may continue to execute. A controller subtree is dead if there are no non-controller processes within it and its nested controller subtree (if it exists) is also dead. Since the implementation base of all objects is self-controlling processes, this definition ultimately reflects the state of primitive processes.

To see how death and abdication are related, consider again Figure 2. If P1 dies, C2 can restore it. If C2 dies, domain D2 is removed, leaving C1 an ordinary shell. The controller privileges of C2 are removed. Since D1 now includes the contents of C1, P1 may restore C2 and reinvest it as the controller of C1. This will recreate the domain D2. If both P1 and C2 die, domain D1 is removed, leaving S0 an ordinary shell and the controller privileges of C1 are removed.

For security and protection reasons, it is unacceptable to allow a deposition or an accidental controller death to reveal the internal structure of an object that wishes to remain entirely abstract. Therefore, HPC allows controllers to choose between two actions to be taken by the system automatically upon abdication, whether forced, voluntary or accidental. One option is to **preserve** all the internal structure that exists at the time of abdication and make this structure part of the enclosing domain. The other option is to **purge** all internal structure before destroying the domain boundary. Using the purge option with images that specify level 1 controllers, arbitrarily complex objects can be created that are indistinguishable from primitive processes to outside controllers.

To summarize the relationships among the operations that manipulate processes and controllers, **animate** implies an **invest**, **abdicate** always implies the last specified of **purge** or **preserve**, **depose** forces an **abdicate**, **die** implies an **abdicate**, and **kill** forces a **die** with the **purge** option.

4. Structured Communication Interfaces

A single unidirectional unreliable channel between two objects is clearly a minimal communications mechanism. HPC provides a number of structuring tools to construct more specialized mechanisms. Provision is made for several logically distinct interfaces, the use of a variety of protocols or data representations, a dynamically varying number of related interfaces, multicasting, and the encapsulation of a number of related, heterogeneous interfaces as a single abstract interface.

Each interface has three fixed properties that are recorded by the HPC kernel and available to the controller: *role*, *type name*, and *structure*. Two interfaces can be connected by a channel only if they have compatible roles, equivalent types, and matching structures. Role compatibility and type equivalence are checked by the individual controller responsible for establishing the connection. Structural matching is checked by the kernel because it has a direct effect on the kernel implementation of message passing. Both roles and types, however, are higher level concepts that are user-defined and, therefore, must be checked at a higher level.

In addition, there are two properties of interfaces that vary: *connectivity* and *liveness*. These properties allow controllers to do a more sophisticated job of managing an application than they could in the absence of the information.

4.1. Roles

Each interface has some particular function or purpose. Since objects must be interconnected in ways that make sense, HPC provides a way for controllers to determine the logical *role* that each interface plays in the operation of its containing object. Examples of roles include the Unix notions of **stdin** (standard input), **stdout** (standard output), and **stderr** (standard error).

HPC records a label for each interface that is made available to the corresponding controller(s), but does not interpret the label in any way. This label indicates the role of the associated interface. Our intent is that object designers will use human-sensible strings for role labels, but there are no hard restrictions on their use.

Role compatibility imposes a user-defined semantic interpretation on interface connections. Each controller is free to interpret role labels as it sees fit and is not required to understand any particular role. Obviously, the more roles a controller understands, the better it will do its job of creating and maintaining complex applications. One or more conventions for interpreting roles are to be expected so that controllers can be written without prior knowledge about specific roles. As an example, a trivial controller could be written that would only connect interfaces with roles of the form **my-input** to interfaces with roles of the form **my-output**. Such a controller need not know anything about the application, only the strings **input** and **output**.

A software development system that supports strong typing and separate compilation might generate distinct roles for each interface and use only controllers that validate roles against a database before establishing connections between objects. Another good use for roles and separate interfaces would be to distinguish the various entry points of an object accepting remote procedure calls or Ada-style rendezvous.

Role names are unique among the interfaces of an object even though there may be cases where there are no discernible role differences, for example, the two inputs in a merge routine. In most such cases, the controller will impose a deterministic schedule that suffices to differentiate the roles. In those cases where the roles are truly equivalent, a multiplex interface (described below) is probably appropriate.

4.2. Type Names

While not strictly necessary, we distinguish between the logical role of an interface and the concrete interpretation of messages sent or received on an interface. Separating roles and types maintains the independence of communication content and communication protocols. A controller can connect two interfaces with compatible roles and equal types without interpreting the particular type. This would be more inconvenient if the labels for the role and type were merged into a single piece of descriptive information.

As with roles, HPC records a *type name* for each interface that is interpreted only by controllers. The type name indicates the protocols and formats used in communication through the interface. **Self-describing datagram** and **reliable bytestream** are examples of type names.

Again, no controller is obligated to understand any particular type name. Simple controllers can check type names for strict equality to ensure that messages sent by one object will be correctly interpreted by another. More sophisticated controllers can interpose protocol or representation translation objects between the objects that are to be connected. For example, a reliable transport protocol object can be

interposed by a controller between two communicating objects that otherwise would use an unreliable communication protocol (see Section 7). Such translation is completely transparent to the objects involved and depends crucially on the type name information being available to controllers.

As for roles, a sophisticated software development system can exploit the type name information to make available at run-time a detailed description of the language and run-time dependent message types, remote procedure call arguments and return types, and so forth. This can be used by controllers to help ensure the sensible interconnection of objects. Type name information is also useful when debugging because it indicates the proper interpretation of messages intercepted at any given interface.

4.3. Structures

For most interfaces, the only relevant structure is the direction of communication (input or output). It is possible, however, to construct more complicated interfaces. Structural matching ensures that inputs are connected to outputs and that previously connected interfaces are not doubly connected.

Unlike roles and type names, the *structure* of an interface is interpreted by the HPC kernel. Interfaces may be connected only when they have exactly complementary structures. The structure of an interface is what HPC uses to determine how to deliver messages.

4.3.1. Endpoints and Extensions

For each of the four interface structures, simple, bundle, multiplex, and multicast, there is a concrete version where the internal details of the structure can be examined and an abstract version where these details are hidden from view. Each view of an interface may independently present either the concrete or the abstract version. A view presenting a concrete version of a structure is an *endpoint*, while an abstract view is an *extension*. The names are motivated because a useful communication link will have two endpoints separated by some number of channels extending the connection between them.

Extensions provide a reduction in complexity and an important improvement in security. A controller may make a connection of arbitrary complexity between two objects without having to cope with that complexity. We can ensure that all necessary logical connections are made with a single operation. At the same time, the status of the extension views can be reported simply, in terms of the entire structure and independently of the states of its individual components.

The endpoint/extension distinction is analogous to the shell/domain distinction for protection purposes. Only the controllers that need to know the internal structure of a complex interface are allowed to observe it or modify it. The operations that are legal for endpoints are not legal for extensions and vice versa. The operations on endpoints all modify the internal structure or state of a complete communication path. This information should be hidden as unnecessary complexity from the domains through which the connection passes. The operations on extensions all modify channels between two interfaces without regard to their type or internal structure and are inappropriate for use on endpoints.

Endpoints differ from extensions in another way. At some point the HPC notion of interface has to be integrated into run-time support for primitive processes. It is essential for at least simple endpoints to receive such support, since processes are the terminals that exchange messages. The view of an interface within a process (like a file descriptor or a similar structure) should be an endpoint. Extensions, on the other hand, make little sense within a process, because they can be connected only to other extensions by channels and we have no intention of extending HPC structuring to within a single process.

Distinguishing endpoints of various structures facilitates the implementation of message transport. As is discussed in section 8.3, messages are not physically forwarded over every channel and through every

interface. Instead, HPC maintains enough information about the channels joined at interfaces to recognize when *end-to-end connections* have been established. Subsequent operations at the endpoints, like sending a message, may be dispatched directly to the opposing endpoint. Most operations on endpoints will have no effect if there is no end-to-end connection. The state of an interface, described below, also depends on an end-to-end connection.

Let us stress the distinction between a channel, a *chain*, and an end-to-end connection. A channel connects exactly two views and crosses no shell boundaries. One or more channels which are joined through interfaces form a chain. When the interfaces at the ends of a chain are both endpoints, there is an end-to-end connection. A controller can directly observe only channels. HPC maintains chains to determine when connecting a new channel will create an end-to-end connection, but chains are completely invisible to controllers. An end-to-end connection can be observed only indirectly, through the liveness property of an interface.

4.3.2. Simple Interfaces

A *simple interface*, consisting of a view that accepts messages and a view that delivers them, provides a single directed stream of messages. The direction of a simple interface is specified when the interface and its containing shell are created. All the more complex interfaces are ultimately based on simple interfaces.

The **send**, **receive**, and **select** operations may be applied to the endpoints, but not the extensions, of simple interfaces. The intent is that messages may be inserted or removed from a stream only at the ends. Besides introducing simplicity, this allows an HPC implementation to reduce the cost of message transport for connections with many channels joined end-to-end to the cost of a single point-to-point transfer because messages may never be observed at any of the intervening interfaces (extensions). We place a further restriction on these operations. They are intended only for use by primitive processes on endpoints within that process and will be provided by the run-time support for HPC. In particular, we provide no way for a controller to **send** a message through an endpoint that is part of a shell that currently has no process animated in it.

Send and **receive** are both non-blocking and transfer messages by value between an interface and a process. They place no interpretation on the contents of a message. **Select** takes a set of interfaces and blocks until a message is available for receipt on at least one of the interfaces or a specified timeout has expired. The reason for unblocking is returned to the process that executed **select**.

The **connect** and **disconnect** operations may be applied to the extensions, but not the endpoints, of simple interfaces. To **connect** two simple interfaces, the views being joined by a channel must have complementary directions.

4.3.3. Interface Bundles

An *interface bundle* is a collection of interfaces, analogous to a record in programming languages. The components of a bundle may be interfaces of arbitrary structure and direction. Bundles can be used to provide bidirectional communication in a single connection by bundling two simple interfaces of opposite direction. Complex protocols that require out of band data could be implemented using bundles of several interfaces to separate message streams. Objects with logically complex interfaces can encapsulate the complexity and group simple interfaces by function.

The components of a bundle are fixed when the interface and its containing shell are created. The order in which they are specified is significant. When two bundles are connected, the corresponding components are connected. The correspondence is determined by the order of definition. This leads to a

simple construction with some inconveniences. While bundles are an obvious choice for bidirectional communication links, the structures at both ends can not be identical. That is, the simple interface for input must come first on one end and last on the other end. We do not have enough experience to evaluate the extent of this inconvenience. We suspect there is already asymmetry in most object relationships and that the inability to provide completely symmetric connections is not a serious drawback.

Figure 3 illustrates this difficulty. In Figure 3, a wide gray line indicates a complex channel and dashed lines indicate *subchannels* connected to the components of the complex interface. The component interfaces of a complex interface are drawn as small interface boxes directly on the body of the complex interface. The shells containing the interfaces in this and subsequent figures are drawn as bold lines separating the two views.

The **index** operation is common to the three complex interface structures. It takes an endpoint and an index and returns the interface component that corresponds to the index (analogous to record selection). This allows HPC to distinguish the properties of the entire bundle from the properties of a particular component of the bundle. The **index** operation may be applied to the endpoints, but not the extensions, of interface bundles. The intent is to restrict any access to the internal structure of a complex connection to the endpoints.

The **connect** and **disconnect** operations may be applied to the extensions, but not the endpoints, of interface bundles. To **connect** two interface bundles, the views being joined by a channel must have the same number of components and the structures of each of the components are compared for compatibility as though they were being connected individually. Interfaces in a bundle are connected only when an end-to-end connection is established between two bundle endpoints. All components of one endpoint are then given end-to-end connections with the corresponding components of the other endpoint. Disconnecting a pair of interface bundles, disconnects all the components.

Even though an interface bundle is analogous to the record as a structuring tool, the indices for bundle are not names, as is common for specifying record fields. Since each component of a bundle carries a role, it should be straightforward for a controller to determine which numeric index should be used to select an appropriate component.

While we describe the **index** operation as returning an interface, it is not necessary to use **index** before each operation on an interface in a bundle. **Index** is simply a way to obtain the names or handles of the component interfaces, which would otherwise not be presented to a controller. Once the name of a particular component interface is known to a controller, it is not necessary to use **index** again to find that name.

The **index** operation should not be thought of as creating interfaces. All the component interfaces exist, just as all the fields of a record exist, but we abstract away that level of detail until it is explicitly requested. In a language run-time package that supports interface bundles directly, the programmer should be able to specify bundle components by name and never deal with the **index** operation directly.

4.3.4. Multiplex Interfaces

A *multiplex interface* is analogous to an array or, more precisely, a table with dynamically varying indices and homogeneous components. A multiplex interface begins with no components. The role, type name, and structure of all its potential components are fixed when the multiplex interface and its containing shell are created. Component interfaces are created and deleted dynamically. This is the type of interface HPC uses for multiplexed servers, hence the name.

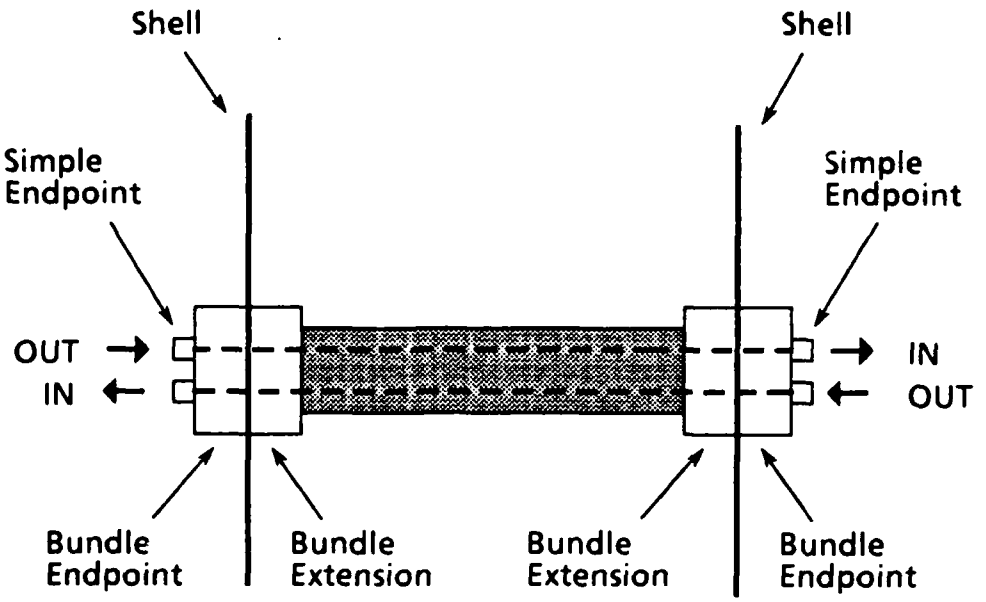

Shell
Shell
Simple
Endpoint
Simple
Endpoint
OUT
IN
IN
OUT
Bundle
Endpoint
Bundle
Extension
Bundle
Extension
Bundle
Endpoint

Figure 3

The **index**, **new**, and **delete** operations may be applied to the endpoints, but not the extensions, of multiplex interfaces. The **index** operation behaves similarly for multiplex interfaces as for interface bundles. However, the valid indices for a multiplex interface change dynamically.

The **new** operation takes a multiplex endpoint and creates a new component interface. The controller is notified that the new interface has been created. If a connection exists between this endpoint and some other elsewhere in the system, as shown in Figure 4a, a new component interface is created for the second endpoint and an end-to-end subchannel is established between the two new interfaces as shown in Figure 4b. The controller of the second endpoint is notified about the (externally initiated) creation of a new interface. If the multiplex endpoint does not have an end-to-end connection with another endpoint, the new component interface is created but no subchannel connections are established for it, as shown in Figure 4d.

The **delete** operation takes a multiplex endpoint and an index and destroys the corresponding component interface. Any connections associated with this component are unmade. If the component has an end-to-end connection, as shown in Figure 4b, and the component is **deleted**, the subchannel between the components is removed but the component on the second endpoint is *not* destroyed. The result is shown in Figure 4c. As will be described below, the controller for the second endpoint can detect the loss of the connection to its component and **delete** the component interface independently. If the component interface on the right in Figure 4c is **deleted**, we return to the situation shown in Figure 4a.

Connect and **disconnect** may be applied to the extension, but not the endpoints, of a multiplex interface. To **connect** two multiplex interfaces, the views being joined by a channel must have compatible component structures. Component structures are compared as though one of each were being connected. Disconnecting a pair of multiplex interfaces disconnects any connected components.

When a **connect** establishes an end-to-end connection between multiplex endpoints, any component interfaces of those endpoints that were previously connected end-to-end as shown in Figure 4b are reconnected. Due to the restrictions on when **new** and **connect** create connections between component interfaces, subchannels of a multiplex interface have a memory of their partners despite disconnections of the entire interface and will only recognize those partners. This is analogous to the correspondence rule for components of interface bundles. Again, we chose a simple rule that provides the necessary function with minimal, but acknowledged, inconvenience.

The primary inconvenience with our correspondence rule for multiplex components is what we call the *tap problem* for debugging or monitoring multiplex channels. One would like to transparently interpose a monitoring object along a multiplex end-to-end connection that could tap all the subchannels of the connection. HPC lacks the ability to do this for two reasons. First, when a multiplex connection is disconnected and reconnected to a different endpoint the currently existing subchannels are inaccessible, even at the new endpoint. Second, although the monitoring object can readily keep track of all the new subchannels created at either of the endpoints during the monitoring session and create corresponding subchannels to the other endpoint to forward messages, there is no way to remove the monitoring object and join the corresponding pairs of subchannels end-to-end. The tap problem is surmountable if one is willing to disrupt ongoing communication between two partners when a tap is inserted or removed, or if one leaves the tap in place over the life of the connection being monitored. We note here that **connect** could be extended in a straightforward way to allow a controller to specify which subchannels currently terminated at a multiplex extension are to be joined to one another. However, all of the particular versions we have examined lead to severe consistency problems.

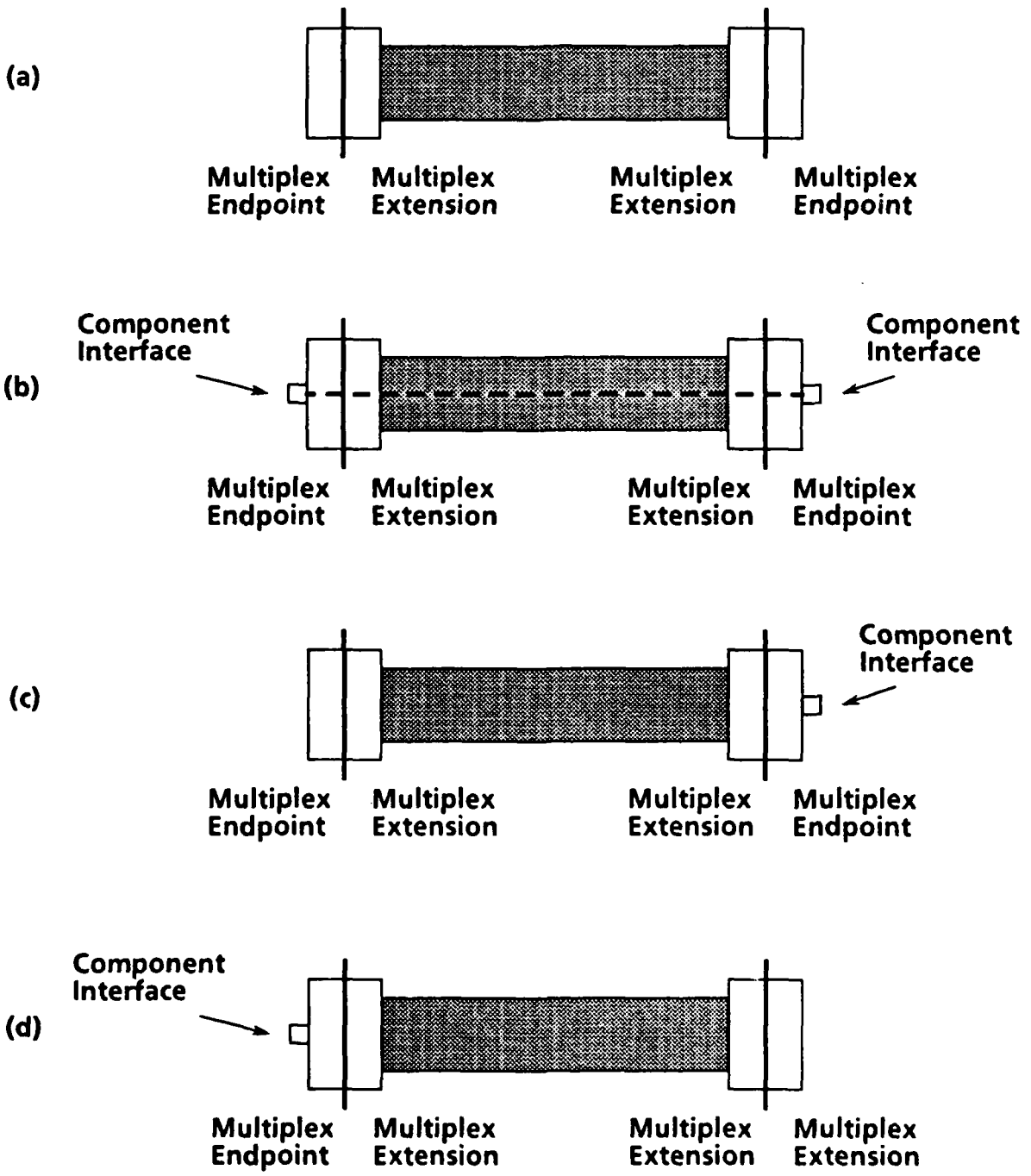

(a)
Multiplex Endpoint
Multiplex Extension
Multiplex Extension
Multiplex Endpoint
(b)
Component Interface
Component Interface
Multiplex Endpoint
Multiplex Extension
Multiplex Extension
Multiplex Endpoint
(c)
Component Interface
Multiplex Endpoint
Multiplex Extension
Multiplex Extension
Multiplex Endpoint
(d)
Component Interface
Multiplex Endpoint
Multiplex Extension
Multiplex Extension
Multiplex Extension

Figure 4

4.3.5. Multicast Interfaces

A *multicast interface* fulfills the need for one-to-many and many-to-one communications in a different way than a multiplex interface. A multicast interface takes several streams of messages, merges them into a single stream, replicates the stream, and forwards it to several destinations. Depending on how much of this function is used, a multicast interface can provide one-to-many, many-to-many, and many-to-one message delivery. Like a multiplex interface, the multicast interface is created with no components and all its dynamically created components have the same fixed structure.

Unlike interface bundles and multiplex interfaces, multicast interfaces do not need to be used in pairs. The extension view of a multicast interface has the same structure as an extension of one of its components. Multicast interfaces *could* always be paired without a loss of function, but this would be inconvenient in the many-to-one and one-to-many cases. It would also be impossible to provide reliable services through replication transparently to clients.

The **index**, **new**, and **delete** operations may be applied to the endpoints, but not the extensions, of a multicast interface. These operations perform similarly as for multiplex interfaces. The major difference is that there is only one connection from the endpoint to another destination; there are no subchannels. All the components of the endpoint share this connection. Messages arriving at any of the components are sent out over the shared connection. Messages arriving on the shared connection are replicated and one copy is sent out through each of the component interfaces.

The **connect** and **disconnect** operations may be applied to the extensions, but not the endpoints, of a multicast interface. As mentioned, the extension view of a multicast interface is an extension with the structure of a component of the endpoint. Thus, a multicast interface with an endpoint view and an extension view serves for one-to-many (see Figure 5a) and many-to-one (see Figure 5b) communication. Two multicast endpoints, either the two views of a single interface (see Figure 5c) or connected by an arbitrarily long end-to-end connection, serve for many-to-many communication.

4.4. States

Besides its fixed role, type name, and structure, an interface has two properties that can change dynamically and unexpectedly.

4.4.1. Connectivity

Connectivity is simply the presence or absence of connections on the visible view(s) of a simple interface and the currently valid indices for a complex interface. Since multiple controller processes in a single controller subtree may create or destroy channels independently of one another and the **new** operation may create new indices for a multiplex interface from another domain, the controller subtree is notified about changes in the connectivity of interfaces in its domain. In all cases, connectivity for a view is based only on visible status.

In the absence of inconsistencies created by partition and merge, the connectivity of a simple interface is either *disconnected* or *connected to I*, where *I* is another interface. The connectivity of a complex interface is the set of currently valid indices.

4.4.2. Liveness

Liveness is the only property controllers can observe about conditions outside their domains. Liveness indicates whether or not an interface is part of an end-to-end connection. A view of an interface is *dead* when no endpoint is connected to the interface through its other view, *alive* when an endpoint is

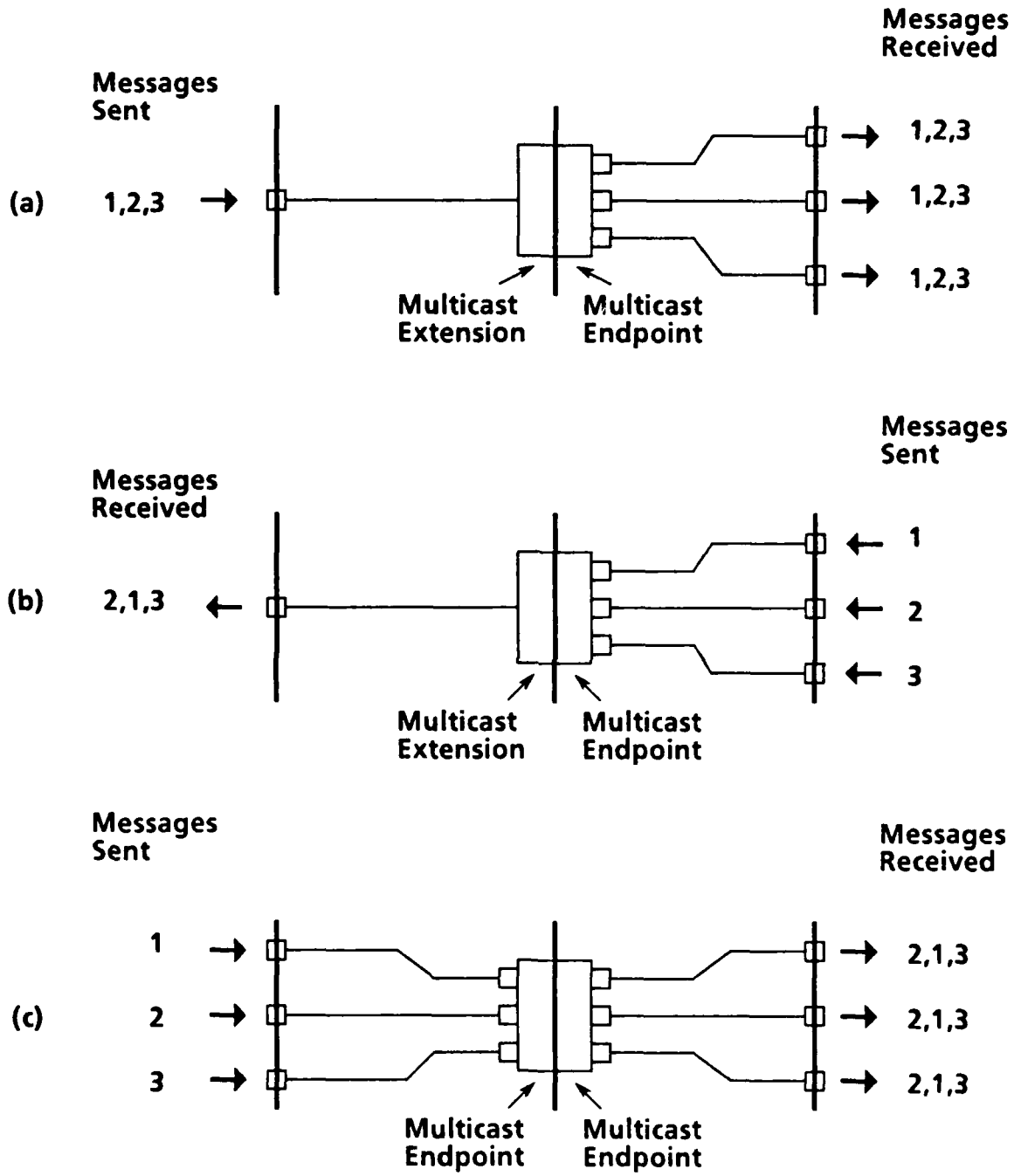

Messages Received
Messages Sent
(a)
1,2,3
1,2,3
1,2,3
1,2,3
Multicast Extension
Multicast Endpoint
Messages Sent
Messages Received
(b)
2,1,3
1
2
3
Multicast Extension
Multicast Endpoint
Messages Sent
Messages Received
(c)
1
2
3
2,1,3
2,1,3
2,1,3
Multicast Endpoint
Multicast Endpoint

Figure 5

connected, and *suspended* when it is unknown whether or not an endpoint is connected. As for connectivity, when the liveness of an interface changes, the controller is notified.

Knowledge of liveness is essential for distributed applications that wish to avoid inconsistency by limiting their operations during a partition. It is also crucial in allowing a controller to clean up state when an external client has died or become disconnected. Basic flow control functions require liveness or its equivalent to detect when a message may be sent with some expectation of being delivered. Applications that are sensitive of their security can use liveness to detect when their communication partners have been disconnected and potentially hostile partners connected.

Liveness information is computed locally at isolated interfaces and the ends of chains and propagated towards the other end of a chain. One view of an interface is dead if the opposite view is a disconnected extension, alive if the opposite view is an endpoint, and suspended if the opposite view is not visible due to partition. If the opposite view is a connected endpoint, this view inherits the liveness of the (third) view at the other end of the channel. These rules are illustrated in Figure 6a. There is one exception to the rules as just given: If a simple endpoint is contained in an empty shell, its opposite view is dead. A simple endpoint must be within a primitive process for its opposite view to be alive.

In Figure 6b, we show more complex examples of how liveness is inherited. Note that liveness of complex interfaces is computed independently of the liveness of their component interfaces and that different components may have differing liveness properties. As a consequence of the behavior of complex interfaces, when a complex interface is dead all its components must be dead. Similarly, when it is suspended all its components must be suspended. However, when it is alive, each of its components may have a different liveness.

4.5. A Comparison with Links

It is appropriate at this point to compare the apparent complexity of HPC's communication structures with the mechanism variously known as *links* or *ports*. In a typical link-based system, a link is a medium of reliable unidirectional communication with access control and naming provided by capabilities. All interprocess communication is accomplished by sending messages along links. The rights to send or receive messages on a link are protected and may be transferred from one process to another.

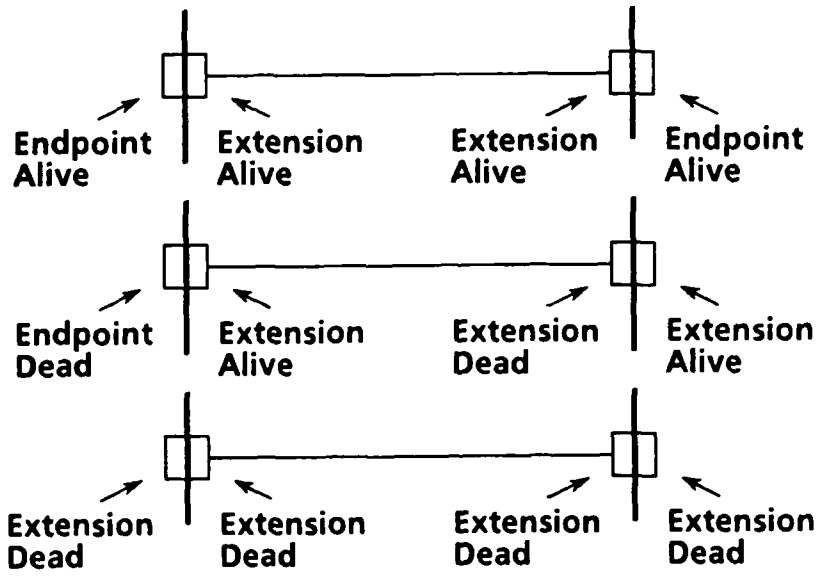
Most link-based systems have been heavily influenced by the Demos[6] operating system. Some, like Demos/MP[13], Perseus[14], Arachne[15], and Charlotte[16] are fairly direct descendants of Demos. Others, like Accent[9] are less directly derived, but share many features with close relatives of Demos.

Link-based systems are currently quite successful and popular. Some of their popularity can be attributed to the apparent simplicity and power of their single communication mechanism. To allow two processes to communicate, one simply creates a link and gives the rights to each end to the two processes. Why does HPC provide so many different communication structures? Why should a designer deal with the additional complexity of HPC communication?

Our response is that the interrelationships among elements of a distributed application are *not* simple. The apparent simplicity of links arises because links hide all the interactions. HPC does not add any additional complexity. It does make the complexity explicit. Moreover, the way various elements are combined into a complex application are visible not just when the system is designed, but also at run-time. In a link-based system, these interrelationships are simply inaccessible to anyone wishing to create an object resembling a controller.

To make the comparison of communication structures a little fairer to link-based systems, consider HPC without nesting of objects. That is, assume that we have only primitive processes in a flat space. Two

(a)


Endpoint Alive
Extension Alive
Extension Alive
Endpoint Alive
Endpoint Dead
Extension Alive
Extension Dead
Extension Alive
Extension Dead
Extension Dead
Extension Dead
Extension Dead

(b)

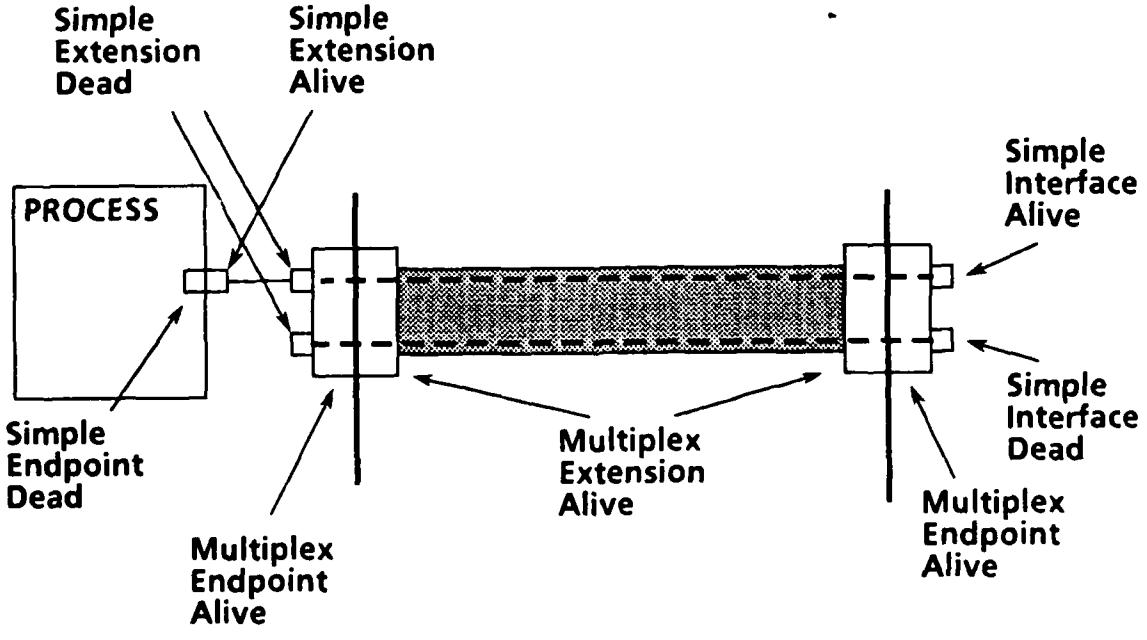

Simple Extension Dead
Simple Extension Alive
PROCESS
Simple Interface Alive
Simple Endpoint Dead
Multiplex Extension Alive
Simple Interface Dead
Multiplex Endpoint Alive
Multiplex Endpoint Alive

Figure 6

simple interfaces and a channel in HPC are the immediate competitors of the link. In HPC, we can see whether or not two processes are connected by a channel. We can take several processes and transparently connect them into a useful assembly. When a process fails, we can replace it and connect its former partners to its replacement without their active cooperation. We can dynamically restructure or extend an application to meet changing demands by creating or moving worker processes transparently. In a link-based system, we can not do *any* of these things because of the way links and all operations on them are handled through capabilities. Only the processes that hold rights to a link know that the link exists. Even these processes do not know what other processes hold rights to that link. There is no way to determine what the relationships among several processes are, even when they all have links with one another.

Now consider interface bundles. Link-based systems provide no direct support for grouping related links. Obviously, each process can keep track of which links belong to which group, but groups of links can not be created, transferred, or destroyed as a group. Each link in a group must be manipulated individually. HPC *reduces* the amount of complexity that a designer must handle by giving support to the abstraction of grouping communication paths at the operating system level.

A multiplexed server in a link-based system normally makes a single *request* link globally available. Clients wishing to set up communications with the server obtain send rights to the server's request link, create a new link of their own, and pass the send rights to the new link in a message over the server's request link. The server will create a third link and send its send rights in a message over the client's link. The client will then release the rights to the server's request link. Subsequent communication between the client and the server will take place over the two newly created links. HPC's multiplex interfaces make this kind of interaction much simpler. A server uses a multiplex endpoint. The components of the multiplex interface are bundles of two simple interfaces. A client creates a new component and connects to the new interface bundle with a single **connect**. The server is notified when the new interface has been created and may connect to it whenever it choses.

Most link-based systems do not have any provision for multicasting or broadcasting. This facility is necessary for reliable applications based on replication. The multicast interfaces of HPC provide this ability and integrate the control and management of multicast groups closely with the rest of the communication structures. Moreover, a client using an internally replicated server need not know that multicasting is involved. A multicast extension is just an extension of one of its components, so a multicast group is addressed in HPC transparently, as though it were a single element. The choice of explicit message replication or multicasting hardware is hidden inside the HPC kernel.

The structured interfaces of HPC do not introduce any complexity into programming distributed applications. They do force the designer to specify the complexities in a system. In return, HPC permits the dynamic observation and manipulation of interprocess communication paths. The structured interfaces of HPC also provide direct support for abstracting away the complexities in a system. The interactions at each level of a system can be specified without having to worry about the levels above and below.

5. Consistency and Reliability

5.1. The Nature of Distributed Consistency

Observations about the structure, state, and behavior of a system are inherently relative to the observer. Changes in one section of a partitioned system will not be visible to all observers. This is the *principle of local observability*. We will assume that all observers at a given site have the same physical constraints on what they may observe, although we will place additional constraints on them for reasons of

abstraction or security.

Change, partition and sharing are the major ingredients in the recipe for inconsistency. In the absence of change, all observations are trivially consistent and trivial to accomplish. Any feature of the system that is immutable may be known uniformly throughout the system without the need for physical observation. The operating system is typically such a feature.

In the absence of partition, all observations may be made consistent in a straightforward fashion by globally simulating a single site. The problems of ensuring consistency by providing serializability or atomicity are understood. Partition is a more serious obstacle to assuring consistency because we have no way to coordinate the operations undertaken in disjoint partitions.

We assume that the topology of each partition is strongly connected, a useful simplifying assumption. While we can not always globally simulate a single site, the property of strong connectivity allows each partition to simulate a single site. The assumption can always be made to hold by simply breaking a partition for which it is not true into individual sites. Whether this is necessary or not depends on the underlying communications topology.

Given that each partition may act as a single site, we can avoid inconsistencies within a partition. This confines the problem of detecting and resolving inconsistencies to the procedures carried out when partitions are merged.

In the absence of sharing, all observations are trivially consistent because only one observer may look at a given piece of the system. In HPC, and other systems, sharing occurs in a number of places and fashions. There are three major forms of sharing in HPC:

(1) Each controller subtree has potentially many principal processes, all of which share control over the members of a domain.

(2) There is explicit sharing of some objects among several domains (see Section 6).

(3) There is implicit sharing of shells and interfaces between the domains on either side of a domain boundary.

We have not yet explored the consistency consequences of explicit sharing. It is clear that explicit sharing has all the same consistency problems as multiple principal processes and implicitly shared interfaces, as well as its own characteristic difficulties. Hopefully, the solutions for the more constrained problems will be easily extensible to the more general problem.

In our view, global consistency in a distributed system is a myth. Either the system is really a centralized system or global inconsistency is inevitable. Global consistency requires that changes only take place when they may be globally observed. While we can prohibit voluntary operations (and thus change) during a partition, we can not prohibit failure. Even if we disregard failures, to ensure global consistency, the complete loss of site autonomy is required. Furthermore, system availability is less than the availability of any of its components. Such a system has all the weaknesses of its components, rather than their strengths. A more realistic and certainly more interesting question is how to live with global inconsistency.

Consider an object as an implementation of some specific abstraction. As long as its internal state is consistent, there is no problem with inconsistent service. If the object is distributed and continues to operate while internally partitioned, the pieces of its state may become inconsistent with one another. This is not yet a problem because each partitioned piece of the object may independently offer service that is consistent over time and over all clients in its partition. However, when the separate partitions are merged, the object can not offer completely consistent service. The object may provide service that is consistent

over time for each client *or* service that is consistent over all clients (in a partition) at any given time. These choices are generally exclusive.

Since we can't specify complete consistency over both space (users) and time in a distributed system, our specifications of the abstractions provided by an object must allow some form of inconsistency. If we insist on consistency over all clients in a partition, we must specify that the object's behavior over time for a given client will be inconsistent. The abstraction implemented by an object is deficient if it does not include a protocol to inform clients of such inconsistencies between past and future behavior. If the protocol consults clients of the abstraction about the preferred reconciliation, so much the better.

The LOCUS file system[17] can be considered to be a distributed application implementing a file abstraction with specific operations and semantics on those operations. Its specification explicitly deals with partition and merge and gives rules for when unequal files are consistent and when they are not. The client is not only notified of the problem, but given sole responsibility for dealing with it. Since LOCUS's internal state is described in terms of i-nodes, version vectors, and disk blocks, inconsistency is discovered in terms of these representations. The abstraction is in terms of files and even during reconciliation the client deals with the abstract files. We stress that the abstraction defined for this distributed application explicitly defines procedures for consulting its clients about inconsistencies and their preferred reconciliation.

Obviously choosing an abstraction that requires the fewest conflicts between space and time consistency reduces the number of times the client must be consulted concerning inconsistencies. Such convenient abstractions can not always be chosen. We may have to meet the demands of clients that do not want to deal with inconsistency. The client pays for such abstractions by lost availability and autonomy.

5.2. The Orphaned Domain Rule

We have discussed how a controller is responsible for managing the internal structure of its domain and how HPC responds to the death of the controller subtree. We now examine how partition of the controller subtree is handled.

We say a domain has been *orphaned* in a given partition when none of the domain's controller processes are in the partition. A primitive process is its own controller, so a primitive process is orphaned in a partition when its site is not in the partition. Obviously, whether or not a domain is seen to be orphaned depends on what partition the observer is in. The controller subtree of a domain may be split among several partitions. As long as at least one controller process is in a partition, the domain is not orphaned in that partition.

All interface views of an orphaned domain are given the suspended state. This state propagates in both directions along any connections through the domain. The effect is to suspend communication with an object during any period in which there is no control component. Another way of describing the same effect is that service is halted whenever there is no responsible party present.

There are a number of motivations for specifying this behavior. First, when a domain is orphaned, all control over the behavior of the domain's members is lost. An application that provides a globally consistent service at the expense of availability during partition must be able to limit the way its components interact with external clients. The orphaned domain rule creates firewalls that prevent the development of inconsistent behavior by prohibiting orphaned domains from interacting with clients in any way.

Second, the orphaned domain rule simplifies many of the rules for maintaining HPC consistency. The example of Section 5.3.1 shows how the number of different kinds of structural inconsistency can be dramatically reduced.

Third, the problem of orphans in transaction systems and RPC is simple to deal with under the orphaned domain rule. If an object becomes orphaned from its controller, either the calling site for RPC or a transaction coordinator, it may continue to perform computations but it will be prohibited from communicating with any other objects. The orphan may be cleaned up by its controller if the controller is still alive when the partitions are merged, or cleaned up automatically by HPC if the controller set the **purge** option.

The strength of the orphaned domain rule is also its major drawback. The members of an orphaned domain may interact with one another even though they are unable to interact with objects outside the domain. This allows useful computation to continue so neither structure or application state need be lost over partitions and merges. However, communication between worker processes inside a domain and their external clients is prohibited even when both ends of an end-to-end connection are within the same partition. We pay for the firewalls of control with potentially lost availability. The appropriate response is to use distributed controller subtrees that have at least one process on every set of sites that can support a viable subset of the application. Given this response, we do not find the rule's drawback more than a minor inconvenience.

5.3. Consistency Of HPC

HPC is intended to run complex applications despite partitions and failures. Clearly HPC is subject to the same cautions and limits as any distributed implementation of an abstraction. In this section we look at two examples of how HPC resolves innocuous inconsistency upon merge and reports the (inevitable) development of conflicts to its clients.

We could have followed the LOCUS example and kept a history of each element of structure (shell, channel, domain, etc) to determine what the dominating, or most recent, version of the system is. We chose to resolve inconsistencies solely on current state without considering past states. The rules we describe here are associative in the sense that any number of partitions may be merged in any order with the same final result. This makes merging more than two partitions at the same time a safe and simple operation.

We also adopted a policy of removing as little as possible from partitions being merged. In the simplest case, two partitions may conflict because an element of structure, say a channel, exists in one partition but not the other. We include the disputed channel in the merger of the partitions. If any previously partitioned part of the system was making use of the channel, it may continue to do so. One consequence is that shells or channels can be deleted in one partition, only to reappear later when another partition is merged with the current one.

5.3.1. Example: HPC Simple Interface State

As discussed in Section 4, the connectivity and liveness observed for each view of an interface may change. Connectivity depends solely on events occurring within the domain to which the view belongs. Liveness depends solely on events outside the domain. When the interface is alive or suspended the HPC kernel maintains additional, unobservable state for the interface. The endpoint associated with the interface is recorded by HPC and used in message transport. This additional state must be kept consistent to assure consistent behavior, even though it is not directly visible to the controller of a domain. In this example we

show how HPC handles consistency for simple interfaces.

When the controller subtree for a view of an interface is partitioned, it is possible that the status of the view will be observed differently in different domains. When merging two (or more) domains HPC must detect and resolve the inconsistency. The method used for detection is not defined by the HPC abstraction, so we will concentrate on reconciliation. We must merge all the changeable state associated with an interface.

Table I shows the possible combinations of connectivity that may be observed in two partitions before and after merging the partitions. **D** indicates the interface is observed to be disconnected. C*i* indicates the interface is observed to be connected with a channel to interface *i*.

Pairs 1 and 2 are completely consistent. Pair 3 is resolved automatically in accord with our policy of least disruption. The last pair is patently inconsistent. When an interface has been connected to different interfaces in different partitions there is no principled way for HPC to decide which channel to remove when merging the partitions. Instead of making an arbitrary choice, HPC reports the inconsistency to the controller subtree of the domain in which the problem occurs. All connections through the interface are forced to the suspended state until the controller resolves the inconsistency. The interface can be fixed simply by removing channels until at most one channel is connected to the interface. Each time a channel is removed the merge rules can be reapplied to obtain the new state of the interface.

Table II shows the possible combinations of liveness that may be observed in two partitions before and after merging the partitions. **D** indicates the interface is observed to be dead. A*i* indicates the interface is observed to be active with a connection to endpoint *i*. S*i* indicates the interface is observed to be suspended with a connection to endpoint *i*, which may be null if no end-to-end connection existed prior to partition.

Pairs 1, 2, and 3 are completely consistent. Pairs 4, 5, and 6 are automatically resolved in accord with our policy of least disruption. We note here that Pair 3 will be observed only for complex endpoints. It is not possible for messages sent in two disjoint partitions to arrive at the same simple (physical) endpoint. For Pairs 5 and 6 we note that if it was possible to reach endpoint *i* in one partition, it will be possible to reach *i* in the merger of the two partitions.

Pairs 7, 8, and 9 are all inconsistent. We do not report these inconsistencies to the controller of the observed domain because they will either be resolved elsewhere in the object hierarchy or will not result in detectably inconsistent behavior and can be ignored.

Table I

Merging Connectivity Observations			
Pair	Before Merge		After Merge
	Partition 1	Partition 2	
1	D	D	D
2	C*i*	C*i*	C*i*
3	D	C*i*	C*i*
4	C*i*	C*j*	Inconsistent

Table II

Merging Liveness Observations			
Pair	Before Merge		After Merge
	Partition 1	Partition 2	
1	D	D	D
2	S*i*	S*i*	S*i*
3	A*i*	A*i*	A*i*
4	D	S*i*	S*i*
5	D	A*i*	A*i*
6	S*i*	A*i*	A*i*
7	A*i*	A*j*	Case 1
8	S*i*	A*j*	Case 2
9	S*i*	S*j*	Case 3

To resolve the inconsistency in Pair 7, we consider how this state must have been reached. In each of two partitions, there existed a connection through the interface to a distinct reachable endpoint. Both endpoints will be reachable in the merger of the partitions. By the orphaned domain rule, each channel participating in the connections is visible to a controller. Since the two connections have disjoint endpoints, yet share at least one interface, there must be a connect-connect inconsistency (viz. Table I, Pair 4) visible somewhere within the merger of the partitions. This inconsistency will automatically suspend both connections until it has been reconciled at the point of conflict. The controller will observe the interface as suspended and HPC may safely make an arbitrary choice between the two endpoints. If HPC chooses the "wrong" endpoint, this will be automatically corrected when the connect-connect conflict is reconciled and one of the connections is allowed to become active again.

Resolution of Pair 8 is similar. Again, invoking the orphaned domain rule, either there is a connect-connect inconsistency as in case 1, or there is a visible connect-disconnect inconsistency (viz. Table I, Pair 3) within the merger of the partitions which will be automatically reconciled by HPC. When the latter situation is resolved, Pair 8 collapses to Pair 5.

In the case of Pair 9, either there is a visible conflict, as in cases 1 and 2, or the point at which the two connections to disjoint endpoints diverge is not visible in this new partition. In this last situation, the controller observes the interface as suspended and HPC may again make a safe arbitrary choice between the endpoints because neither of the endpoints can be observed in the new partition. Choosing the "wrong" endpoint will be detected and resolved as soon as enough information to resolve the conflict has been merged with this partition.

Under this specification, consistent behavior is guaranteed by forcing an interface to the safe suspended state. In only one case is a controller required to deal with an inconsistency, and all other potential inconsistencies of behavior have been either reduced to an instance of this single case elsewhere in the hierarchy or made innocuous by forced suspension of the interface.

5.3.2. Example: HPC Object Hierarchy

HPC ensures a consistent tree structure of shells, within a single domain and partition. However, when the controller subtree of a domain is partitioned, the internal hierarchy of the domain may be

modified inconsistently. Upon merge, a consistent hierarchy must be reestablished. In this example we will look only at inconsistencies that affect a single domain. Every object (in the absence of explicit sharing) is supposed to have one parent. An object gets a new parent when the surrounding shell is **disclosed** or it is **enclosed** by a new shell. The controller of a domain may observe the parentage of any object in its domain.

We can maintain consistency largely because the operations on shells are so limited. In particular, there are no general tree manipulation commands, only the **enclose** and **disclose** operations. Each enclose operation creates a new, uniquely named shell.

Table III shows the possible combinations of parentage that may be observed in two partitions before and after merging the partitions. N indicates the shell is not observed. C*i* indicates the shell is observed to be a child of shell *i*.

Pairs 1 and 2 are completely consistent, although pair 1 would probably never be considered during a merge. Pair 3 is resolved in accord with our policy of least disruption. Pair 4 is inconsistent. The controller is notified of this inconsistency. Until it is resolved, all connections passing through the object are suspended and the only operations that may be applied to the object are **kill**, **depose**, **disclose** and a special reconciliation operation, **detach**, that is used to remove the object from the tree of a specified parent. An object may be detached from a parent only when it has more than one parent.

Figure 7a depicts a parentage inconsistency. Object C was **enclosed** by shells P1 and P2 in two different partitions. When the partitions were merged, the conflict became apparent. If we **detach** C from P2, we are left with the situation shown in Figure 7b. The **detach** operation is somewhat provisional. Further development of explicit sharing of HPC objects may suggest a more general replacement.

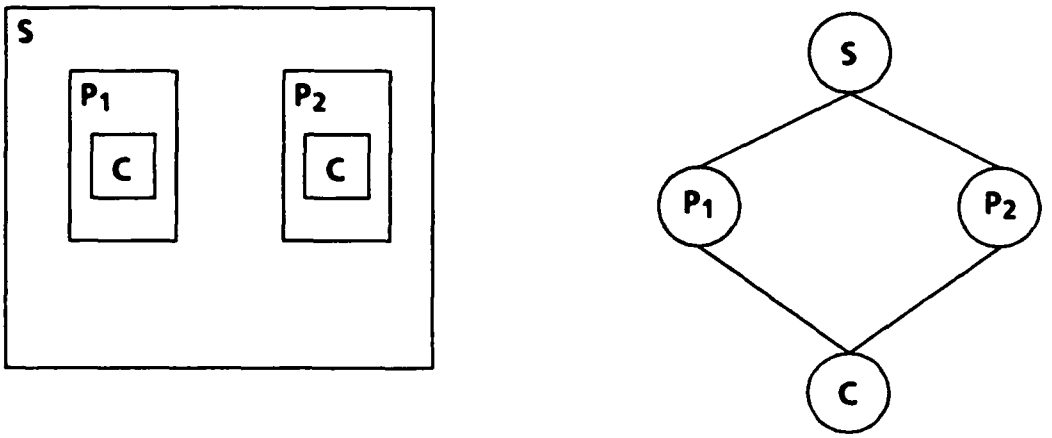
When a shell with a parentage conflict is **disclosed**, all its children inherit its parentage conflict. This may multiply the number of conflicts but there does not seem to be a reasonable alternative. If **disclose** is prohibited, we are unable to separate the contents of an object according to the partition and role they were created in and reencapsulate them. The need for such fission can be avoided by never creating anything in a shell that existed before the controller subtree was partitioned, but this runs counter to the basic HPC approach of high availability even during partition.

5.4. Provisions for Reliable, Consistent Applications

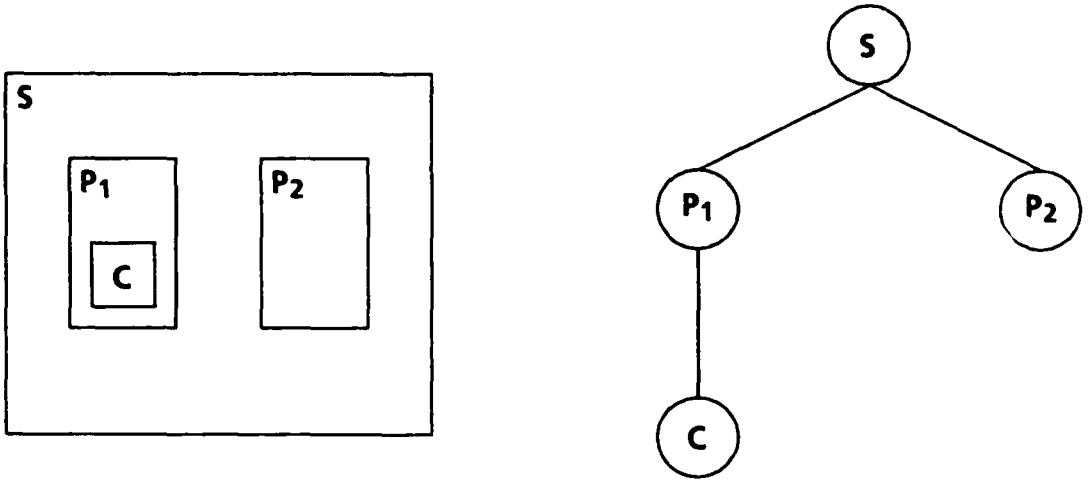
HPC provides support for reliable applications in the areas of communications, replication, and distribution. The design obligation was not to provide high-level reliability mechanisms like transactions,

Table III

Merging Parentage Observations			
Pair	Before Merge		After Merge
	Partition 1	Partition 2	
1	N	N	N
2	C*i*	C*i*	C*i*
3	N	C*i*	C*i*
4	C*i*	C*j*	Inconsistent



(a)



(b)

Figure 7

checkpoints, or triple modular redundancy directly, but rather to provide the tools to build these somewhat specialized mechanisms. Since HPC makes partition visible to controllers and all resources can be encapsulated as an HPC object, there is also basic support for building consistent applications.

Applications that must be robust against single point failure require some form of broadcast or multicast mechanism. If there is no primitive broadcast, either the kernel or a user process may simulate multicast by replicating messages and sending them to all the individual members of the multicast group. This works but ordinarily requires clients of reliable applications to be aware of the internal organization of the application. Ideally, a multicast group would look just like an ordinary destination to a client. In HPC we achieve tight protection and a clean separation between multiple delivery and addressing through multicast interfaces. The kernel takes care of any necessary replication.

One can easily construct in HPC the troupe mechanism described by Cooper[18], which uses replicated groups of both client and server processes with every process in one group sending a message to every process in the other group. Each troupe may be implemented as an HPC object with a multicast interface. Components of the multicast endpoint inside the shell would be connected to the members of the troupe. Two troupes would be connected simply by connecting the multicast extensions outside their shells. Each troupe would have a similar interface for communication with the ringmaster. The ringmaster would have a multiplex interface to communicate with the dynamically varying number of troupes and each component of its multiplex interface would be a multicast extension. More restricted mechanisms, like triple modular redundancy, are even easier to construct.

Figure 8 illustrates two troupes and the ringmaster. We omit the controller details and the internals of the ringmaster for simplicity. Notice that troupes all have similar external interfaces, no matter how many replicated members they have. Similarly, each member of a given troupe has similar external interfaces. The complexity of replicating members of a troupe is isolated within the shell of the troupe, while the complexity of a variable number of troupes and their interconnection is isolated at the next higher level. Troupes can be built independently of their use and used independently of their construction.

Physical distribution is another mechanism needed by reliable applications. HPC does not distinguish among objects on the basis of their location. In fact, only primitive processes can be said to have a location in HPC. HPC places almost all physical resource management responsibilities in the hands of ordinary HPC objects and neither hinders nor helps a designer spread an application over several sites. The placement of primitive processes is limited solely by the reachable resource managers. Unlike Argus guardians[19], the processes contained in an HPC object may be placed on any site. HPC is also neutral regarding stable storage. Stable storage may be encapsulated by an HPC object like any other resource.

Implementing principals as controller subtrees allows a designer to exploit physical distribution of control as well as computation. In fact, the orphaned domain rule encourages the use of multiple controller processes, one for each site which contains some portion of the application. If multiple controller processes were not allowed, applications would be subject to severe availability penalties under the orphan rule and would be subject to single point failure, no matter how widely distributed they might be.

The HPC kernel checkpoints various aspects of structure to assist in recovering failed objects. For each interface the kernel records its role, type and structure. For each shell that has been animated, the kernel records the arguments most recently used to create a process in it. For each shell that is or was a domain, the previous controller subtree is recorded. Each of these features helps a newly invested controller know what the functions of each element of its domain are and recover failed elements. As an example, in the case of controller death and automatic abdication with the preserve option, the controller that inherits the domain can determine which objects were part of the controller subtree, create

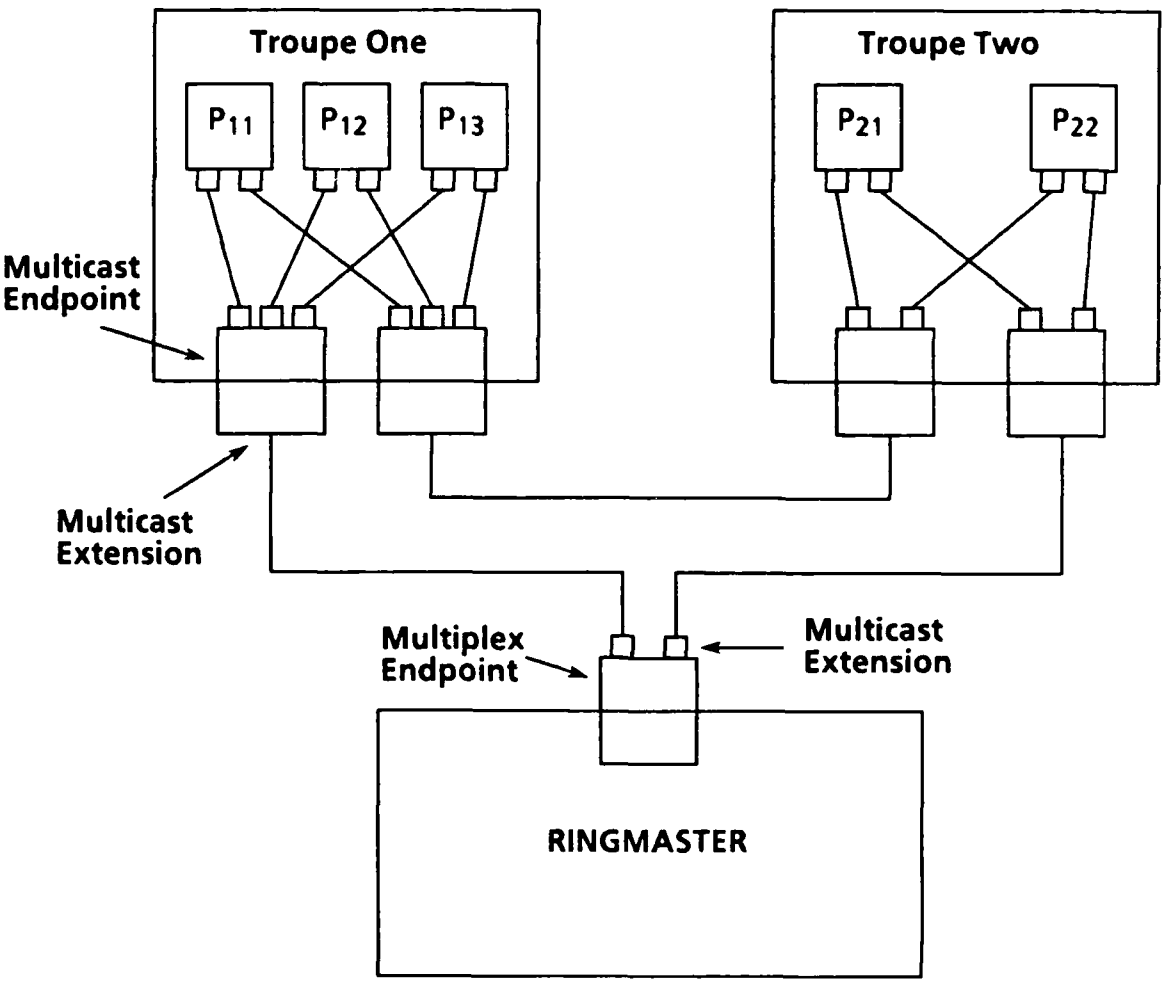

Troupe One
Troupe Two
P11
P12
P13
P21
P22
Multicast
Endpoint
Multicast
Extension
Multiplex
Endpoint
Multicast
Extension
RINGMASTER

Figure 8

replacement processes in their shells, and reinvest the controller subtree as the principal for the inherited domain.

The indication of partition and failure represented by changes in the liveness status of interfaces allows applications to take a wide variety of consistency, recovery, and general reconfiguration actions. Transaction coordinators can use suspended status and timeouts to build a system tolerant of partition. Applications that require a number of different types of worker objects can create additional workers to replace those lost through failure or partition and dynamically insert them into the computation. Servers can detect when communication with their clients has failed and clean up state accordingly.

Applications can be easily written that configure themselves according to the demands for their services by maintaining just enough worker objects to meet the demand. If the protocol for requesting service includes information about where the user is physically (an optimization), the application can change not only the size but also the location of its implementation base.

6. Sharing

One of the primary responsibilities of an operating system is to allow multiple users to share resources. In HPC, objects are used to represent resources. All requests for service use the standard form of inter-object communication, messages sent to interfaces bound with channels. The problem with this view is that a request for service may come from deep within a user's object hierarchy and be directed to a resource that resides deep within the system's object hierarchy. In order to implement such a request, each operation domain would have to propagate an appropriate interface from the leaf nodes of the object tree to the root. In addition, since some controller must establish communication connections, this suggests a *root controller* must be responsible for creating a connection between the operation domains at the root of each subtree.

This approach is impractical for two reasons. First, it suggests that every operation domain residing between two communicating objects in the object hierarchy must be involved in all communication between the two objects. Even if interface connections are implemented so that messages are not actually forwarded through passive interfaces in shells, some machinery must be invoked within each operation domain to check interface compatibility and to maintain the appearance of local control. Secondly, the code required within each controller to make connections for all communication between objects in its subtree with objects within other subtrees would be considerable, greatly increasing the complexity of each controller. An alternative approach is to have two subtrees share an object that can communicate directly with each of the subtrees. We will describe a provisional mechanism for sharing in HPC; a refinement of this scheme is currently under investigation.

One of the fundamental motivations for the sharing mechanism is the desire to provide a clean relationship between clients and servers when their closest common ancestor is many levels above in the hierarchy. Sharing is simply a form of communication that obviates the need to describe all communication between subtrees in terms of connections through the root of the object tree. We have chosen to share objects rather than channels or interfaces. Although the shared object presents itself in two domains, only a subset of its interfaces are visible in each domain. Thus, shared objects are an exception to a strict interpretation of the rule that every object resides within a single domain, although each interface of a shared object does obey the rule.

Sharing of an object is accomplished by negotiation between the two domains involved. There are two steps necessary to share an object: (1) the controller of the domain containing the object to be shared must **offer** to share the object with another specified domain, naming those interfaces to be made available

to the other domain's controller and (2) the controller of the domain in which the object is to be introduced must **accept** the object and a subset of its interfaces.

Offer and **accept** are system calls that do not require a connection between the two domains. However, if two domains wish to negotiate, the controllers may use any user-defined protocol for this purpose. All negotiations take place using messages sent through channels; only the final result is implemented by the kernel. Most instances of sharing will not require negotiation, hence, there is no need for the corresponding controllers to communicate directly. A system switchboard can be constructed using a domain that offers to share its interfaces with the world. This is a special case of **offer** that allows global services to be provided.

An offer can be made at any time; an acceptance will succeed only if a corresponding offer has been previously made. When a controller makes an offer to share an object, it loses any rights it would otherwise lose if the offer were immediately accepted. In particular, any connections extant at the time of the offer to interfaces named in the offer are immediately broken. Therefore, there is no intermediate step during which two controllers have access to the same interfaces.

The net result of this sequence of operations is that a single object now resides within two domains, although the two controllers are constrained in their ability to interfere with each other. Each controller is capable of establishing connections with a nonoverlapping subset of the object's interfaces. Normally, a controller has the right to connect or disconnect channels to any interface associated with an object within its domain. However, if an object is shared, only a subset of interfaces remain within the controller's domain. The offer and acceptance identify which interfaces are to be controlled by each domain.

Sharing naturally extends to more than two domains. Each domain can share a subset of the interfaces it controls with another domain, which in turn may share some subset with a third. Also, moving an object from one domain to another is just a special case of sharing. A domain can offer all of an object's interfaces to another domain, which, if accepted, will have the effect of moving the object to the new domain.

We should note that the naming operations necessary to share an object do not impose an additional burden on either controller. The controller offering to share an object within its domain clearly knows the names of its interfaces. Similarly, a controller that accepts an object must know the names of those interfaces it is to control in order to make use of them.

7. Applications

In this section, we will describe how to use the HPC model to build representative system-level applications. Our goal is to show that our underlying assumptions are not too restrictive and that the model is sufficiently powerful to describe typical system services.

7.1. Reliable Communication

Since the HPC model is intended to support software at very low levels in the system, it is natural to assume the minimum functionality, namely asynchronous, unreliable communication. This is clearly inadequate for most user-level programs, however. Fortunately, HPC provides a framework for building higher-level communication protocols.

The interface between a user-level program and a reliable protocol can be either procedure-based or object-based. We will describe an object-based scheme similar to that used for network protocols; a procedure-based approach is independent of the HPC model and will not be discussed. Each machine

provides a *reliable transport object* (RTO) to be interposed between a sender and receiver that require a reliable channel for communication. The RTO is responsible for implementing the appropriate protocol, possibly including checksums, acknowledgements, and retransmissions. The local RTO communicates with remote RTO's to implement reliable remote communication.

Inserting protocol objects between two communicating objects can be viewed as type coercion by the controller. In Figure 9, two objects on different machines want to communicate. Each has specified, using type names *wants-reliable-input* (WRi) and *wants-reliable-output* (WRo), that they would like reliable communication. The controller can interpose two RTO's that communicate with each other using a reliable protocol, represented by type names *reliable-input* (Ri) and *reliable-output* (Ro). The effect is to transparently coerce an inherently unreliable channel (the communication medium) into a reliable one. The two processes involved have no way of knowing that intermediate objects are processing all messages.

This solution assumes reliable communication between objects and their local RTO. The end-to-end argument[20] implies that this is not strictly necessary, since higher-level checking will be required anyway. In addition, we do not want to require that the model provide reliable communication for objects on the same machine because it would limit implementation choices in the kernel (*e.g.*, the kernel could not choose to ignore a message when there is no buffer space) and also undermine the transparency of machine boundaries in the model. Reliable local communication can be accomplished by using a procedure interface and a shared data structure for communication with the protocol object.

The RTO is an example of a machine-specific binding of an object. Each machine environment provides a shared RTO that acts much like any other service, offering a multiplexed interface to the various users. Each machine has an RTO and no RTO is allowed to migrate to another machine. When a process wishes to communicate reliably, it must be connected to a reliable transport object on the same machine. This connection, as well as the location of the RTO, is transparent to the communicating process, but not to the associated controller.

7.2. Debugging

By nature, distributed programs are very difficult to debug due to asynchronous execution and communication delays. The most common approach to debugging distributed programs is to provide mechanisms for monitoring communication between processes. Each process can be viewed as a sequential program to which traditional debugging techniques can be applied. The interactions between processes are captured in messages, which can be monitored or changed by the underlying support system. This approach has the advantage that it can be made transparent to the program and is language independent. However, a debugger that can only monitor message traffic and other primitive events is of limited use. We see an obvious need for support for additional levels of abstraction, those defined by user-level software, during the debugging process. To avoid redefining abstractions during debugging, the system must provide a general framework for constructing abstract levels within distributed programs and make them available during the debugging process. Three advantages to using the HPC model for this purpose are: (1) the model is general, subsuming all abstract layers of a program, including most operating system functions, (2) the model explicitly supports abstraction, both during program construction and during execution, and (3) the model provides a framework for transparent monitoring of system-defined operations, as well as higher-level, user-defined functions.

The most important goal of the HPC model is to provide support for abstract objects in a distributed system. This support takes the form of system calls to create objects, connect communicating objects, and protect object abstractions. These calls result in modifications to the object hierarchy, a data structure

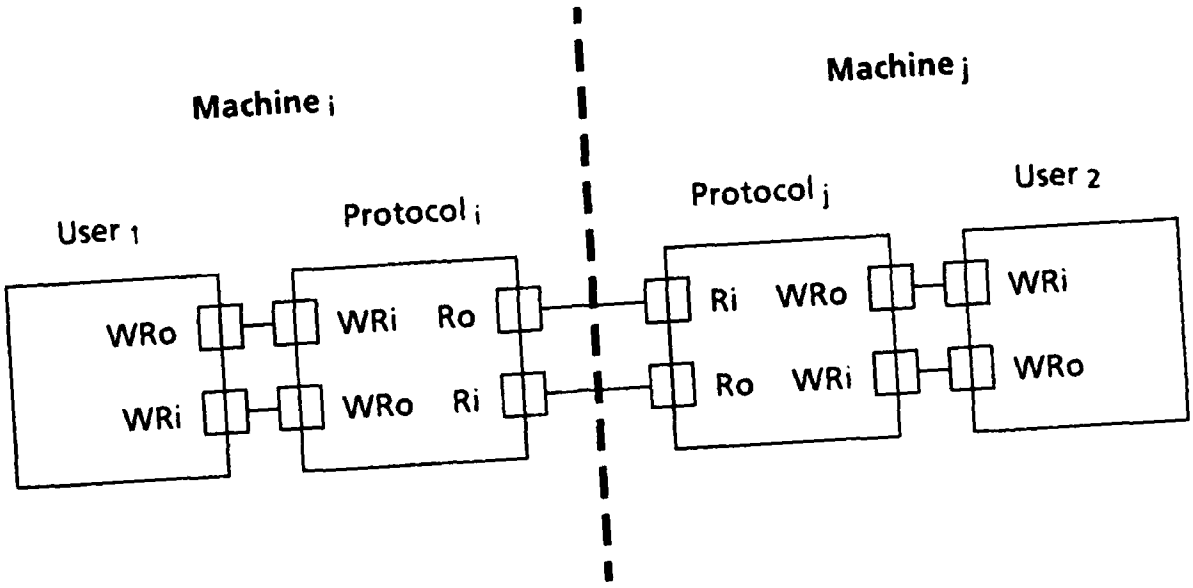

Machine i
Machine j
User 1
Protocol i
Protocol j
User 2
WRo
WRi
WRi
Ro
Ri
WRo
WRi
WRo
Ri
WRo
Ro
WRi
WRi

Figure 9

maintained by the HPC kernel at run-time which serves as a concrete representation of the user's conceptual model of the program. The relationship between objects, *as expressed by the program*, is present at run-time and can be exploited by the debugger.

There are two different ways to transparently monitor HPC message communication depending on the assumptions under which the program was constructed. In many user-level programs, no special control is required and, therefore, a general system-defined controller is used for communication connections. This *universal controller* (analogous to a shell in Unix) would implement normal connections, but would also provide the capability of monitoring all connections it helps to establish. In particular, a debugger process could be interposed between every pair of communicating processes by the universal controller.

An alternative approach would be used if the user program required a controller tailored to the specific application. In this case, the user-defined controller would have to perform the functions of the system-defined controller, as well as those additional operations needed by the user. This approach is not truly transparent to the user program, since the controller, which is a part of the program, must make the appropriate connections for the debugger. However, the controller is independent of the *computation* within the program and is used solely for control. The communicating processes are unaware that messages are being monitored.

Controllers could also be used to monitor abstract states of interest to the programmer. A controller is often required to know something about the abstract state of the objects within its domain in order to maintain the consistency of the domain abstraction. This abstract state can be passed from the controller to higher-level mechanisms, including controllers higher in the object hierarchy and the debugger, making it possible for the user to monitor changes in abstract state during the debugging process. This is especially attractive because it limits the amount of information presented to the user depending on the current focus of interest. In this way, the debugger can provide "telescoping" of abstractions, that is, the ability to monitor events at any level in the object hierarchy.

Finally, HPC provides the ability to control abstract objects directly. In particular, it is possible to "freeze" an entire object consisting of multiple communicating processes. This makes it easier to control the order of events during execution because, as with message monitoring, it limits the scope of concern. Rather than forcing the user to impose a specific schedule on all processes in the system, only the objects of concern are affected.

7.3. Lightweight Transactions

Zwaenepoel and Almes have suggested a technique for implementing lightweight transactions based on temporary output files.[21] In their scheme, each worker process participating in a distributed computation is given a ticket (capability) representing a unique identifier, a set of input files, and a set of output files. Each worker process uses temporary files for output, so as to have no effect on the shared file system prior to a commit. A centralized job manager is responsible for assigning tickets to workers and collecting their results. A temporary file is renamed (atomically) by the manager process to commit the results of a computation. Since a worker's results are written into a temporary file, the actions of a worker process appear atomic. If the manager process is aborted, the rename operation will never be executed, hence the workers will have no permanent effect on the file system. If a worker process is partitioned from the manager, a new worker process can be created by the manager with a new ticket. The results of the previous worker process will never be renamed since only the most recent ticket is allowed to cause a commit. Eventually the orphan worker process will complete or abort; in either case its results will be

discarded.

The problem with this technique is that it has very limited application. It works nicely for the class of programs that predeclare all files to be used during execution and produce whole files as output, but does not apply to programs that dynamically open input and output files. In such a case, the manager process is unable to determine what files will be used by a worker process and, therefore, cannot create a ticket. The coarseness of control appears to be the limiting factor in this scheme. The HPC model provides the necessary fine-grain control needed to implement this form of lightweight transaction.

In the HPC model, a controller process is not simply another user process with greater responsibilities. In HPC, a controller process is responsible for creating and maintaining the structure of its domain. In particular, a controller must establish all communication connections within its domain, including connections to the file system. This is exactly the level of detail required to implement lightweight transactions using atomic file renaming. Since the controller is already interposed between the user process and file system, it is in a position to perform file rename operations, even on dynamically created files.

One problem with this solution is that it doesn't address the relationship between the file name and it's contents, particularly if the mechanism is to support nested lightweight transactions. For example, if a process writes the file's name into the file and some other process is allowed to read it before it is committed, the internal name and external name won't match. A possible solution would be to have the controller monitor every interaction with the file system (that is, every read/write), but this would likely impose an unacceptable amount of overhead. The scheme works best when all files are opened for reading or writing, in which case it provides a very cheap, but useful, form of transaction.

8. Implementation Issues

In this section we discuss some of the issues that arise in the implementation of an HPC kernel. Many related issues are as yet undecided, since an implementation of HPC is just now under development.

8.1. Protection Without Capabilities

One goal of the HPC model is to explore an alternative to capabilities as a protection mechanism. We wish to minimize that portion of a distributed system which must be reliable for the system as a whole to operate. Systems which have either general capabilities or "port rights" as a special case of capabilities must provide reliable mechanisms for their transmission and control. Experience with Unix IPC[22] both at the University of Rochester and Carnegie-Mellon University suggests these mechanisms are difficult to implement, especially across local-area networks. HPC puts the transfer of access rights in the kernel as part of the intrinsic support for protection and structure and makes it unnecessary to support the transmission of capabilities in user message traffic.

Additionally, access and communication control in capability-based systems is potentially chaotic. Even given system support for capabilities to control access, there is no general agreement on how to control the proliferation and distribution of capabilities. In systems where there are mechanisms to control the spread of capabilities, these mechanisms provide a limited set of constraints.[23] To determine potential holders of access rights in a general capability system, one must do an analysis analogous to global data-flow analysis. Even this may be insufficient, given run-time input as a source of communications.

There is an analogy between capabilities and typed, validated pointers in a programming language. Both are useful, but difficult to control and understand in complex structures and often more powerful than is necessary. HPC operation domains are roughly analogous to scopes in programming languages - simple

to identify and control, and generally sufficient to express desirable abstractions. For situations where a pointer-like mechanism is desired, we provide sharing of objects, which is more controlled than general capability replication and which appears adequate for most distributed computations.

8.2. Interface Compatibility

The HPC kernel is responsible for ensuring that two interfaces have matching structures before a connection is allowed. Sufficient information about each interface is maintained within a data structure describing the object tree to allow the kernel to enforce structural matching. Role compatibility is defined by the individual controllers and requires no system support.

The controller is also responsible for ensuring type equivalence of connected interfaces. Type checking of interfaces could be done by any one of a number of techniques. One approach, often used in programming languages, is to statically declare interfaces and connections. This is an unsatisfactory solution for HPC because it would not allow arbitrary dynamic connections. To guarantee type equivalence of interfaces at run-time would be rather expensive, in that dynamic type checking would be necessary. We will proceed instead with a simplified type checking scheme using type names to determine equivalence. Each interface is given a type name upon creation; two interfaces are type equivalent if the type names match. (Note this is not the same as *name equivalence* in programming languages.) Name forgery is not a concern because only authorized controllers are capable of making connections; we are primarily interested in catching errors. Since the type name is meaningless to the system, designers are free to create them and interpret them any way they like. To increase confidence in the typing system, user code could be compiled in an all-encompassing separate compilation system, but the underlying system would be oblivious to it.

8.3. Forwarding Messages

In most message-based systems, message interfaces are associated with active processes. In HPC, shells have interfaces: processes are simply a special case of a shell. All messages are eventually interpreted by processes, but may have to travel through several logical interfaces before reaching an active destination. To actually forward messages through all the logical "gateways" between two terminal processes presents an unacceptable burden. Experiments with functional abstraction in the RIG distributed operating system[24] were hindered by the inability to separate the abstract interface (gateway) from a physical process and port. Fortunately, there is a simple way to obtain the efficiency of end-to-end message passing with the structure of HPC and its logical gateways.

The HPC kernel maintains a record of the logical connections between interfaces. For a series of channels between active processes, the kernel records both the logical path and the active endpoints. Messages, which can only originate at one of the active endpoints, are immediately sent to the other endpoint, logically passing through multiple interfaces, even though only one physical channel is used. However, disconnecting a channel between any two interfaces on the path between two active components breaks the physical connection. This approach allows each controller to maintain control of its connections, while supporting an efficient implementation of message passing.

To monitor or modify messages on a channel passing through multiple interfaces, a controller can insert a process into the logical path by breaking an incoming connection, inserting a connection to the new process, and creating a new outgoing connection. This action is transparent to the processes at the endpoints. The effect is to break the previous logical path into two logical paths, each sharing the new "tap" process as a terminal and requiring no special treatment.

8.4. Domain Protection

The operations that modify a domain can only be issued by the controller of the domain. In order to check the validity of an operation domain command, the object hierarchy is maintained as a tree with four fields in each node containing the name of: (1) the object represented by the subtree, (2) the domain inside the object, (3) the domain outside the object, and (4) the domain controlled by the object. Every node will contain at least the first three names, although some subtrees will not have an entry in the last field, since not all subtrees are part of a controller. These fields are initialized when the object is created; they may be updated each time an operation domain is created or destroyed. In particular, if a controller contains a sub-object that is made the controller of a nested domain, that sub-object is no longer allowed to control the external domain.

To determine whether or not a domain modification operation is valid, we must determine whether the operation arguments are in the same domain and whether the process issuing the call is in the controller for that domain. For example, consider an object X that issues a call to connect an interface in object Y to an interface in object Z. To check the legality of the call, we first determine whether Y and Z are siblings or whether one is the parent of the other, a necessary condition for a direct connection. If a connection is possible, we can determine whether X is a controller for the corresponding domain using the appropriate field in the object tree. Note that this information could be calculated from the relative positions of object X, Y, and Z in the object tree, but this would only improve the efficiency of domain creation, while introducing overhead for each domain modification operation. We expect operation domain creations and deletions to be much less frequent than structure modifications within a domain.

9. Conclusions

One theme of this work is that *operating systems facilitate or obstruct solutions to problems in much the same way that programming languages do.* In distributed systems we are forced, almost of necessity, to build more complex systems to take advantage of the intrinsic properties of distribution. The underlying operating system should make it straightforward to develop and implement such systems. The HPC model is a step towards such a system. We believe that the desired features can be made efficient enough to satisfy all but a small number of distributed applications. We have emphasized the utility of services rather than their raw performance.

If one wishes to support a special class of applications, such as distributed databases or transaction systems, the operating system can be chosen to provide services of tremendous specific utility. The HPC model has built-in biases and assumptions about the way applications will be designed. We assume that hierarchies will be the most common structure and that the implementation (and thus specific processes) of a given service should be hidden from the client. The intent is to provide mechanisms sufficiently general to let the user build whatever application structures are desired and avoid special-purpose mechanisms. Given several alternatives, we have chosen the one that: (1) makes the fewest assumptions, (2) can be used to implement the others most precisely, (3) is most independent of other aspects of the system design, and (4) can be used to solve several problems.

HPC was designed to support the construction of distributed operating systems, as well as other support layers. It is not intended to be a user presentation layer, witness the fact that communication in HPC is asynchronous and unreliable, an unattractive combination for user building blocks. It is important to note, however, that higher level abstractions, such as atomic transactions and reliable protocols, can be built using the HPC mechanisms.

A major assumption of this work is that interesting applications have significant, long-lived, internal communications patterns and that most applications will not be multiplex servers with well-known addresses. To acknowledge and manipulate the relationships between processes, connections should be named and distinguished. The primary objections to connection-based communications have been in the cost of establishing and destroying connections. The assumption of long-lived multiple-process structures makes the argument of overhead cost less convincing. Further, there is no constraint on the implementation to provide an abstract service based on connections in terms of physical connections; a connectionless implementation might serve just as well. The user should see logical connections between communicating elements regardless of how the transport protocols are implemented.

Our design principles may be summarized as follows:

- Hierarchical process structuring based on abstraction and composition
- Asynchronous message passing over persistent connections
- Dynamic system configuration under user control
- Controlled access without capabilities
- Little dependence on particular resource management facilities
- Emphasis on the tools needed to build reliable mechanisms

We do not expect the HPC model to be a panacea for structuring operating systems. The benefits of the model have an associated cost that may be impractical for some operating system functions. The full costs of establishing and maintaining the relationships implied by the model have yet to be determined, but will surely have an impact on its applicability. The suitability of imposing the HPC model on the interactions between two processes will depend on their frequency of execution, granularity of interaction, relative failure probabilities, and the level of error recovery required. In particular, we do not expect significant advantage will be gained by imposing the model on extensive single-site interactions, such as that between a paging process and a disk device process. For reasons of consistency and transparency, we believe it will be necessary for the model to support all relationships, regardless of machine boundaries. However, the model is most useful for describing long-lived relationships that span multiple machines. For such long-lived, distributed computations, the HPC model provides a uniform framework for all system services, including communication, protection, resource management, and error recovery.

References

1. A.K. Jones, The Object Model: A Conceptual Tool for Structuring Software, in *Operating Systems - An Advanced Course*, R. Bayer, R.M. Graham and G. Seegmueller (ed.), Springer Verlag, New York, 1979, 7-16.
2. P.M. Schwarz and A.Z. Spector, Synchronizing Shared Abstract Types, CMU-CS-83-163, Carnegie-Mellon University, Pittsburgh, Nov 1983.
3. C. Hewitt and R. Atkinson, Parallelism and Synchronization in Actor Systems, *Proceedings 4th Symp. Principles of Programming Languages*, Los Angeles, Jan 1977, 267-280.
4. A. Goldberg and Robson, *Smalltalk-80, The Language and Its Implementation*, Addison-Wesley, New York, 1983.
5. S.J. Leffler, W.N. Joy and M.K. McKusick, UNIX Programmer's Manual, 4.2 Berkeley Software Distribution, Computer Science Department, University of California at Berkeley, August 1983.
6. F. Baskett, J.H. Howard and J.T. Montague, Task Communication in Demos, *Proc. 6th Symp. on Operating Systems Principles*, West Lafayette, Indiana, Nov 1977, 23-31.
7. M.H. Solomon and R.A. Finkel, The Roscoe Distributed Operating System, *Proc. 7th Symp. on Operating System Principles*, Pacific Grove, Calif., Dec 1979, 108-114.
8. C.S. Carr, S.D. Crocker and V.G. Cerf, Host-Host Communication Protocol in the ARPA Network, *Proceedings AFIPS Spring Joint Computer Conference 36*, (May 1970), 589-597.
9. R.F. Rashid and G.G. Robertson, Accent: A Communication Oriented Network Operating System Kernel, *Proc. 8th Symposium on Operating System Principles*, Pacific Grove, Calif., Dec 1981, 64-75.
10. G. Almes, A. Black, E.D. Lazowska and J.D. Noe, The Eden System: A Technical Review, Technical Report 83-10-05, Dept. of Computer Science, University of Washington, Oct 1983.
11. E. Lazowska, H. Levy, G. Almes, M. Fischer, R. Fowler and S. Vestal, The Architecture of the Eden System, *Proc. 8th Symp. on Operating Systems Principles*, Pacific Grove, Calif., Dec 1981, 148-159.
12. C.S. Ellis, J.A. Feldman and J.E. Heliotis, Language Constructs and Support Systems for Distributed Computing, Technical Report 102, Department of Computer Science, University of Rochester, May 1982.
13. M.L. Powell and B.P. Miller, Process Migration in DEMOS/MP, *Proc. 9th Symp. on Operating Systems Principles*, Bretton Woods, N.H., Nov 1983, 110-119.
14. W. Zwaenepoel and K.A. Lantz, Perseus: Retrospective on a Portable Operating System, Technical Report STAN-CS-83-945, Stanford University, Feb 1983.
15. R.A. Finkel and M.H. Solomon, The Arachne Distributed Operating System, Technical Report #439, University of Wisconsin - Madison Computer Sciences, July 1981.
16. Y. Artsy, H-Y. Chang and R. Finkel, Charlotte: Design and Implementation of a Distributed Kernel, Technical Report 554, University of Wisconsin - Madison, August 1984.
17. B. Walker, G. Popek, R. English, C. Kline and G. Thiel, The LOCUS Distributed Operating System, *Proc. 9th Symp. on Operating Systems Principles*, Bretton Woods, N.H., Nov 1983.
18. E.C. Cooper, Circus: A Replicated Procedure Call Facility, *Proceedings 4th Symposium on Reliability in Distributed Software and Database Systems*, Silver Spring, Maryland, 15-17 October 1984, 11-23.

19. B. Liskov and R. Scheifler, Guardians and Actions: Linguistic Support for Robust, Distributed Programs, *ACM Transactions on Programming Languages and Systems 5*, 3 (July 1983), 381-404.

20. J.H. Saltzer, D.P. Reed and D.D. Clark, End-To-End Arguments in System Design, *ACM Transactions on Computer Systems 2*, 4 (Nov 1984), 277-288.

21. G. Almes and W. Zwaenepoel, Understanding and Exploiting Distribution, TR85-12, Department of Computer Science, Rice University, Feb 1985.

22. R.F. Rashid, An Inter-Process Communication Facility for UNIX, Technical Report CMU-CS-80-124, Carnegie-Mellon University, Apr 1981.

23. W.A. Wulf, R. Levin and S.P. Harbison, *Hydra/C.mmp: An Experimental Computer System*, McGraw-Hill, New York, 1981.

24. J.E. Ball, J.A. Feldman, J.R. Low, R. Rashid and P. Rovner, RIG, Rochester's Intelligent Gateway: System Overview, *IEEE Transactions on Software Engineering SE-2*, 4 (1976), 321-328.

END

DTIC

8-86